Broadview
www.broadview.com.cn

Alibaba Group 阿里巴巴集团 | 技术丛书

逆流而上

阿里巴巴技术成长之路

阿里巴巴集团成长集编委会 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

阿里巴巴集团成长集编委会

由阿里巴巴集团不同业务线及不同技术领域内的人员组成的虚拟组织。技术人员都知道软件开发过程中的八二原则，理解大多数问题发生在何处、发生的原因、如何解决，变得尤为重要。阿里巴巴集团业务飞速发展，技术人员积累了大量丰富的线上问题排查及解决的案例和经验。

成长集编委会从中挑选了一些优秀的技术案例，侧重于对问题的还原和分析。我们希望，曾经踩过的坑都能具有其意义和使命，而后来者通过学习前人的经验，防微杜渐，快速成长。

AIS官方微信公众号

阿里技术微信公众号

逆流而上

阿里巴巴技术成长之路

阿里巴巴集团成长集编委会 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是阿里巴巴集团荣耀背后的技术血泪史。全书通过分享业务运行过程中各个领域发生的典型“踩坑”案例，帮助大家快速提升自我及团队协作，学习到宝贵的处理经验及实践方案，为互联网生产系统的稳定共同努力。从基础架构、中间件、数据库、云计算、大数据等技术领域中不断积累经验，颠覆技术瓶颈，不断创新以适应不断增长的需求。

本书主要面向互联网技术从业人员和在校师生，使读者能够通过此书基本了解阿里在各技术领域的能力，学习在如此规模下可能出现的问题以及解决方案的探讨和沉淀分享。

图书在版编目（CIP）数据

逆流而上：阿里巴巴技术成长之路 / 阿里巴巴集团成长集编委会著. —北京：电子工业出版社，2018.1
ISBN 978-7-121-32768-1

Ⅰ. ①逆… Ⅱ. ①阿… Ⅲ. ①软件开发 Ⅳ. ①TP311.52

中国版本图书馆 CIP 数据核字（2017）第 235791 号

策划编辑：董　雪
责任编辑：林瑞和
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：13.75　字数：200 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

本书编委会

编委会成员（排名不分先后）：

曹项、黄愜姣、林跃华、刘雪银、梅庆、钱少琼、沈乘黄、施海燕、施晓霞、王薇娜、吴建玉、周洋

内容撰写（排名不分先后）：

基础架构：林跃华、牛立杰、张学功、赵树起、朱晓波、邹巍

中 间 件：李凯、吕婧、孙棋、王建伟、张乎兴、祝坤荣

数 据 库：董联红、李鹏、廖涵、罗龙九、吴喆赟、章颖强

业务研发：陈庆相、李嘉鹏、任涛、张俊龙、张小菲、朱云锋、邹果、曾薇

运行管理：付来文、吴昌龙、张森扩、赵伟、周洋

特别感谢（排名不分先后）：

TIAN MIN、XU MIN SCHUMANN、蔡晓丹、黄荣刚、金高平、李东旭、李茂凯、李敏、李娅莉、路凯、梅庆、罗玥、强音、荣华、苏奕、孙磊、申建、王子凌、杨华、周鹭、周明、周荣茂对本书出版工作的支持!

自序

2017 年 7 月 27 日，阿里巴巴集团（简称为阿里）市值超过 4040 亿美元，成为亚洲第一。回首过去 18 年的历程，伴随着阿里业务从电商快速成长到覆盖金融、云计算、物流等众多行业的是阿里技术人在基础设施、操作系统、中间件、云等各个领域孜孜不倦的探索、创新和实践。

在每一个技术领域，我们尝试过业务问题多种不同的解法，无论是新技术还是成熟的解决方案，我们都充分验证，直至完全掌握。但在我们看来，最宝贵的并不是我们最终采用某种技术或方案的决定，而是大家在探索中遇到的问题以及解决办法，是对每种技术深入研究过程中积累的经验，是基于对技术深入理解的基础之上进行调优和定制的实践。

随着互联网的浪潮日益高涨，我们看到越来越多的技术人开始经历相似的过程，单纯“拿来主义”的技术方案已经无法满足各个行业层出不穷的业务创新，唯有完全掌握技术才能使之贴合业务需求，更好地服务客户，而掌握技术的关键就在于解决它在实际应用中产生的问题。所以，我们把阿里落地各类技术过程中遇到的问题以及解决方案分享给各位同仁，希望对大家开阔思路、少走弯路能够有所帮助。

本书总结了阿里巴巴集团的技术团队在基础架构、中间件、数据库、业务开发以及运行管理等领域的经典实践，从采用的方案、遇到的问题、

解决方法以及对未来的思考等方面，全面介绍技术实践的细节。在编写方面，本书注重实操，包含代码示例、排查思路及处理流程，以便读者快速应用到自己的工作中。本书也特别邀请了首任的阿里稳定性负责人振飞和周明为本书撰写推荐序，分享他们一路走来的酸甜苦辣。

非常感谢阿里各条业务线的技术同人，在百忙之中安排时间总结、整理并撰写案例，用他们的经验反哺技术同行，这也是阿里技术人为互联网技术不断创新贡献的一点微薄之力。

沈乘黄

阿里基础设施事业群全球运行指挥中心总监

推荐序 1

我从 2009 年 9 月 25 日奉命组建淘宝技术保障部，到 2016 年 4 月 1 日移交 AIS（Alibaba Infrastructure Service）给新任 CTO，历时 2380 天，大约每 3 小时经历一次故障，可以说每天的生活就是从一个故障走向另一个故障，那段日子里我无时无刻不在琢磨如何保障并提升阿里平台的生产稳定性。淘宝/支付宝的可用性从 2009 年的 99.5%到 2010 年的 99.95%，到后来逐年提升并保持到现在的 99.99%，由 AIS 牵头、协同集团各 BU 的技术小二集体为此付出了巨大而卓有成效的努力。从我的视角看，有以下三点经验。

一、做好顶层设计

“不谋全局者，不足谋一域”。生产稳定性的保障不能只埋头于一时一事的细节中，按照马云老师在 2009 年年底对我讲“不仅要救火，更要防火”的要求，必须做好顶层制度设计。

1. 研发和运维团队要能够“向对方靠近并迈一步，互相理解和尊重”，这其中的过程改进（SPI）和配置管理（SCM）的同人可以起到独特的承上启下的作用。这样技术保障部的基本组成是：

SPI + SCM + Production Engineer + DBA + System/Network Engineer

而且团队要逐步加强研发能力，并能够对整个系统架构进行代码级的把控。

2. 故障的标准统一以及处理流程的持续强化。2009 年年底，我们讨论明确淘宝/支付宝的 P1 故障定义为“成交下跌 10%且持续 10 分钟以上”，以此为准绳，统一思想和故障处理应急指挥体系，以及坚持事后故障复盘。事实证明，牵住了这个“牛鼻子”对稳定性工作有很大提升。

3. 坚持建设阿里经济体统一的基础设施平台。AIS 从小变大的过程，就是淘宝、阿里云、B2B、支付宝等技术保障团队逐步融合的过程，也是原本分散的各种软硬件基础设施逐步融合的历程，坚持“书同文、车同轨、行同伦”。没有统一的基础设施和标准规范（包括 IDC、网络、服务器、OS、中间件、数据库、业务应用、研发运维系统及工具、支持 HTTPS 标准等），就根本做不到今天的稳定性。

二、坚持技术创新

阿里巴巴过去 18 年的大发展是业务不断创新的过程，同样，阿里生产系统的稳定性也经历了持续不断的技术创新。

1. 积极推动“去 IOE”和金融级云数据库 OceanBase 的发展及成熟。此创新使得阿里交易和支付系统架构可以灵活地支撑业务飞速发展，技术完全自主可控、积累了众多基础工程技术和人才，也大幅降低了技术成本。

2. “异地多活”和全链路压测。2010 年，我们就开始从青岛机房尝试做淘宝交易的“异地多活”，历经多年的反复技术尝试，终于有了今天北部、中部、南部的多机房同时支撑交易支付的能力。2012 年“双 11”零点惊魂促使我们下决心搞定“全链路压测”，用模拟的流量进行极限压测以获得生产系统的真实负载能力，经过 2013 年、2014 年连续两年的实战摸索，现在已然成为我们“双 11”稳定运行的利器。

3. 云计算技术的逐步应用和强大。2009 年，阿里云正式成立，2012 年，“双 11”天猫电商云平台“聚石塔”首次采用阿里云的产品支撑，到今天云计算在阿里巴巴平台广泛地使用和“云化”，都是咬牙坚持技术创新的结果。

4. 统一计算平台到 ODPS。没有统一的计算平台，不仅会造成技术力量分散且成本不可控，更会导致数据生产和维护的混乱，是稳定性的大患。2014 年启动“登月计划”，打造阿里集团统一的底层大数据平台，满足安全性、可管理、能开放等重要业务需求，在 2015 年 6 月完成了阿里所有数据业务的运行平台从 Hadoop 升级到飞天 ODPS；同时在迁移过程中建立数据管理基本规则，做到业务升级再造和数据通用。

三、组织管理创新

阿里经济体是一个朝气蓬勃的商业生态，一直在持续不断地进行业务创新；背后支撑这个生态的是一个超级复杂的技术体系，运行和维护这个技术体系也需要进行组织管理方面的创新。

1. 设置 PE（Production Engineer，生产工程师）岗位，掌控业务应用的生产维护工作，这个岗位介于业务研发、DBA 和系统及网络工程师之间，起到重要的桥梁纽带作用，为对口各 BU 的业务平稳运行负责。

2. 成立 GOC（Global Operations Center，全球运行指挥中心）、指定生产应急值班长，牵头负责整个阿里经济体技术平台的日常运行和维护。故障的监控、报警、指挥、消防、事后复盘等全流程的运行管理，并通过持续的故障演练保障系统稳定性。特别是 2015 年启动对核心交易和支付系统的“生产突袭”，是一种特别有效、真刀真枪的检验业务生产连续性能力的举措，应该长期坚持做下去。

3. 面对“双 11”的技术保障体系。针对一年一度的天猫全球狂欢节，

日常的保障措施是远远不够的，需要成立单独的技术“团部”掌控全局，各关键链条上的 BU 成立“技术连部”决策局部稳定性，以及精干的“情报分拣中心”担当最辛苦的枢纽，负责判断每条业务线情报员上报的各种异常信息并及时给出动作。

有了顶层设计、技术创新和组织变革，最终落实生产稳定性的还是靠一线技术小二一行行的编码、一次次的测试、日复一日不厌其烦的故障排查工作，以及我们对维护生产稳定性小二们工作的重视、肯定和发自内心的欣赏。他们不是所谓的技术大牛或大 V，不会在各种论坛上侃侃而谈，也不会书写高大上的 PPT；他们面对日常一个个突发的故障，遭受委屈、忍受冤枉、不惧倒霉、坚忍不拔；他们是脚踏实地、埋头苦干的无名英雄，是阿里技术的脊梁。这本书《逆流而上：阿里巴巴技术成长之路》就是负责阿里大平台生产稳定性的部分技术小二的代表，把他们这些年在基础架构、中间件、数据库、业务研发、运行管理等大型互联网平台的稳定性建设中积累的宝贵的实战经验，用平实无华的语言娓娓道来，这些技术沉淀既是对过往典型故障的深度分析，也是跟同行们切磋交流的宝贵知识财富。

我要深深地感谢过往七年里为阿里生产系统稳定性付出努力的所有技术小二，也特别高兴看到本书的出版，并愉快地推荐给所有关心互联网平台稳定性的同行们。

刘振飞

阿里巴巴集团首席风险官（CRO）

原阿里技术保障部（AIS）负责人

推荐序 2

外界对于阿里巴巴技术的了解，要么是“双 11”又创造了交易和支付的世界峰值纪录，要么是阿里云技术的高大上，要么是又出了什么黑科技，非常炫。在这绚丽的背后，有那么一群技术人，是他们支撑了 7×24 小时不间断的 Online 服务，他们付出了艰苦的努力，也踩过了无数的坑，是他们让无数的业务想法变成了现实，感谢在背后默默付出的阿里技术人！

这本《逆流而上：阿里巴巴技术成长之路》从业务运行的角度，收集了不少的实际案例，来自阿里的多个技术团队，内容从第三方的运营商、DNS 到 IDC 机房，服务器、网络到存储，中间件、数据库到业务系统和运行管理等，几乎囊括了运行的所有技术环节，也验证了技术之外的经验“对生产系统保持敬畏之心”“千里之堤，毁于蚁穴”，所有的这些都极具参考价值。

共享是互联网最重要的精神，阿里巴巴技术人希望将这些血和泪的教训分享出来，和技术同人共同成长，如果这些分享能够给同行带来一些共鸣或者启发，那将是阿里技术人最大的幸福！

周明

阿里巴巴集团副总裁

阿里基础设施事业群（AIS）

目录

第1章

基础架构高可用

互联网、电子商务、云计算、大数据等领域都离不开基础设施建设，基础设施的建设包含从服务器到 IDC、网络等一系列的过程。技术日新月异变化的过程，也给基础设施的建设带来了巨大的挑战。如何用研发能力解决基础设施稳定性是基础架构不断提升核心竞争力的关键。

本章从线上案例入手，从基础架构角度介绍了问题排查思路和创新设计概念，希望给读者带来启迪。

1.1 明察秋毫，域名解析排查技巧

背景

网站服务是现在众多互联网服务中应用最广泛的服务之一，而且中国的网络环境更为复杂，几乎每个应用都会使用到域名。域名在互联网应用中是非常重要的一环，万一出现问题可能会导致整个网站无法访问，并且由于各地 DNS（Domain Name System，域名系统）缓存，有时候恢复解析所需时间会比较长。本节针对几种典型的域名解析问题，做了简单的排查思路介绍，以期与读者交流。

域名解析的过程

用户使用终端（电脑、手机、Pad 等）上网时，电脑会设置一组 DNS（即缓存 DNS，又叫 Local DNS）作为一层“代理”、“缓存”，为当地局域网或其他用户提供高性能的解析迭代查询。这个缓存 DNS 一般是在用户连上网时自动分配的，不需要手工配置。当用户访问网站时，如 www.example.com，会向缓存 DNS 发送 example 对应的地址的查询请求。当缓存 DNS 收到请求后，如果其有相应的缓存且没过期，则直接响应；否则，向根 DNS、顶级域 DNS、主域名对应的权威 DNS 或更下级的权威 DNS 进行查询，直到查询到最终的地址，然后返回给客户端。

对网站管理者（更准确地说，是域名的管理者）来说，需要为域名设置解析，将域名指向自己的网络服务器。购买了域名，一般会使用注册商的 DNS（即将注册商提供的 DNS 服务器作为购买的域名的权威 DNS 服务器），或者使用第三方的 DNS。如果使用注册商的 DNS，设置解析的流程

会比较便捷；但如果使用第三方的 DNS，需要在第三方添加域名、解析记录，然后在注册商处修改域名的 DNS 作为第三方提供的 DNS 名字。

在上述描述域名解析的过程中，有两种 DNS 的角色：

- 缓存 DNS，又叫 Local DNS 或 Cache DNS，一般是由网络接入服务提供商在近上网用户端做的缓存 DNS 服务器，主要用于迭代查询的缓存，其中的数据均来自向上级的查询得到，本身并无实际的权威数据，这既增加了解析速度，又减少了对权威 DNS 的查询数量，就好比是数据库中的 Memcache。
- 权威 DNS，又叫 Authority DNS。由域名管理者负责管理，或委托其他 DNS 服务提供商进行管理，其中的记录均是由负责本域的管理员添加进来的。权威 DNS 是由上到下一级级授权的，如根 DNS（ROOT DNS）将各个后缀授权到各个后缀管理机构的顶级域 DNS（即注册局的 DNS），而各个顶级域 DNS 中又有指向下级域对应的 DNS。

如何判断是否是解析异常

网站无法访问有多种原因，如网络异常、服务器异常、网站配置不正确、代理服务器异常等。

当你访问网站时，如果打开的页面内容和预期的不同，或报错信息中含有“resolve”相关字样，或提示“找不到网站服务器”时，就应该怀疑是不是解析方面的问题了。

排查解析问题用到的基本方法是用 dig（或 nslookup，本文以 dig 为例来说明，nslookup 相关命令请参考其对应的帮助文档）看能否正常解析到最终的 IP。

如果无法获取到最终的 IP，或者获取到最终的 IP 不是网站的实际 IP（这个由域名网站管理员判断），就是解析异常了。

如下是一个正常的解析（部分域名用 example.com/bar.com/bar.net 等指代，部分 IP 地址使用 1.1.1.1、2.1.1.1、2.2.2.2 等代替，后续类同）。

```
$ dig www.example.com

; <<>> DiG 9.8.3-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12758
;; flags: qr rd ra; QUERY: 1, ANSWER: 6, AUTHORITY: 0, ADDITIONAL:0

;; QUESTION SECTION:
; www.example.com.          IN   A

;; ANSWER SECTION:
www.example.com.    26   IN   CNAME    www-jp-de-intl.example.com.
www-jp-de-intl.example.com.    165 IN    CNAME
www-jp-de-intl.example.com.gds.bar.com.
www-jp-de-intl.example.com.gds.bar.com. 26 IN A 2.1.1.1

;; Query time: 1 msec
;; SERVER: 1.1.1.1#53(1.1.1.1)
;; WHEN: Mon Apr 24 10:54:46 2017
;; MSG SIZE  rcvd: 210
```

如下场景能获取到解析值，但解析到的 IP 是错误的（具体原因后续分析）。

```
$ dig www.example.com

; <<>> DiG 9.8.3-P1 <<>> www.example.com
```

```
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 8876
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL:0

;; QUESTION SECTION:
; www.example.com.          IN   A

;; ANSWER SECTION:
www.example.com. 1041 IN   A    93.46.8.89

;; Query time: 1 msec
;; SERVER: 1.1.1.1#53(1.1.1.1)
;; WHEN: Mon Apr 24 10:57:26 2017
;; MSG SIZE  rcvd: 50
```

解析异常的排查分析

解析的技术原理，就是域名从根域将顶级域授权给顶级域的权威服务器，再由顶级域的权威服务器授权给主域名的 DNS 权威服务器；查询时，客户端查询 Local DNS，再由 Local DNS 向根域、顶级域、主域名的权威 DNS 查询。如果解析的值是 CNAME 或 NS，则再次查询对应的值，直到获取到 IP 或超时。

排查时，也是按照这个原理来进行的。

正常的迭代查询结果

技术人员可以用 dig 做手工的迭代查询，如下是正常的查询。

```
$ dig www.isc.org +trace
```

```
; <<>> DiG 9.8.3-P1 <<>> www.isc.org +trace
;; global options: +cmd
.           52015   IN   NS   b.root-servers.net.
.           52015   IN   NS   k.root-servers.net.
.           52015   IN   NS   i.root-servers.net.
.           52015   IN   NS   j.root-servers.net.
.           52015   IN   NS   d.root-servers.net.
.           52015   IN   NS   m.root-servers.net.
.           52015   IN   NS   h.root-servers.net.
.           52015   IN   NS   f.root-servers.net.
.           52015   IN   NS   a.root-servers.net.
.           52015   IN   NS   l.root-servers.net.
.           52015   IN   NS   g.root-servers.net.
.           52015   IN   NS   c.root-servers.net.
.           52015   IN   NS   e.root-servers.net.
;; Received 477 bytes from 30.26.8.65#53(30.26.8.65) in 40 ms

org.            172800   IN   NS   b0.org.afilias-nst.org.
org.            172800   IN   NS   b2.org.afilias-nst.org.
org.            172800   IN   NS   d0.org.afilias-nst.org.
org.            172800   IN   NS   c0.org.afilias-nst.info.
org.            172800   IN   NS   a2.org.afilias-nst.info.
org.            172800   IN   NS   a0.org.afilias-nst.info.
;; Received 431 bytes from 192.33.4.12#53(192.33.4.12) in 257 ms

isc.org.        86400    IN   NS   ns.isc.afilias-nst.info.
isc.org.        86400    IN   NS   ams.sns-pb.isc.org.
isc.org.        86400    IN   NS   ord.sns-pb.isc.org.
isc.org.        86400    IN   NS   sfba.sns-pb.isc.org.
;; Received 260 bytes from 199.249.112.1#53(199.249.112.1) in 809 ms

www.isc.org.        60   IN   A    149.20.64.69
```

```
isc.org.        7200 IN   NS   ns.isc.afilias-nst.info.
isc.org.        7200 IN   NS   ord.sns-pb.isc.org.
isc.org.        7200 IN   NS   ams.sns-pb.isc.org.
isc.org.        7200 IN   NS   sfba.sns-pb.isc.org.
;; Received 276 bytes from 149.20.64.3#53(149.20.64.3) in 153 ms
```

域名不存在或域名被 Hold

当域名不存在（如域名输入错误，域名过期被删除等），或者域名存在但被 Hold（如域名未实名认证等，根据政策的不同，可能会有被 ClientHold/ServerHold 的处理。被 Hold 的域名将从顶级域的 DNS 服务器中被删除）。总之，是在顶级域的 DNS 服务器中，把域名向下级授权的 NS 记录（域名服务器记录）删除了，导致查询时无法获取到主域名的 NS 记录。前面看到正常的迭代查询，一级级的是：

. IN NS ROOT_NS

TLD. IN NS TLD_NS

NAME.TLD IN NS DOMAIN_NS

HOST.NAME.TLD IN A IP

而域名不存在的，可以看到查询结果中并没有出现 not-exists-domain.com ** IN NS dns.dnsname.com 这样的记录，在顶级域的 DNS 中返回的是 SOA 记录(如 SOA a.gtld-servers.net. nstld.verisign-grs.com. 1493003321 1800 900 604800 86400)。

不存在域名的 dig 例子。

```
$ dig www.not-exists-domain.com +trace
; <<>> DiG 9.8.3-P1 <<>> www.not-exists-domain.com +trace
;; global options: +cmd
```

```
.           3600 IN   NS   k.root-servers.net.
.           3600 IN   NS   j.root-servers.net.
.           3600 IN   NS   i.root-servers.net.
.           3600 IN   NS   h.root-servers.net.
.           3600 IN   NS   g.root-servers.net.
.           3600 IN   NS   f.root-servers.net.
.           3600 IN   NS   e.root-servers.net.
.           3600 IN   NS   d.root-servers.net.
.           3600 IN   NS   c.root-servers.net.
.           3600 IN   NS   b.root-servers.net.
.           3600 IN   NS   a.root-servers.net.
.           3600 IN   NS   m.root-servers.net.
.           3600 IN   NS   l.root-servers.net.
;; Received 505 bytes from 30.26.8.65#53(30.26.8.65) in 12 ms

com.            172800   IN   NS   e.gtld-servers.net.
com.            172800   IN   NS   c.gtld-servers.net.
com.            172800   IN   NS   f.gtld-servers.net.
com.            172800   IN   NS   l.gtld-servers.net.
com.            172800   IN   NS   b.gtld-servers.net.
com.            172800   IN   NS   j.gtld-servers.net.
com.            172800   IN   NS   i.gtld-servers.net.
com.            172800   IN   NS   m.gtld-servers.net.
com.            172800   IN   NS   h.gtld-servers.net.
com.            172800   IN   NS   k.gtld-servers.net.
com.            172800   IN   NS   a.gtld-servers.net.
com.            172800   IN   NS   d.gtld-servers.net.
com.            172800   IN   NS   g.gtld-servers.net.
;; Received 503 bytes from 192.33.4.12#53(192.33.4.12) in 169 ms
com.            900  IN   SOA  a.gtld-servers.net. nstld.verisign-
grs.com. 1493003321 1800 900 604800 86400
;; Received 116 bytes from 192.48.79.30#53(192.48.79.30) in 204 ms
```

被阻断

正常的迭代查询是根、顶级域、域名的权威 DNS 一级级的 NS 授权，如果被阻断，在应该返回 NS 记录的响应中获取了 A 记录，而中间缺了 DNS 的由上到下的一级或多级的授权 NS 记录，这不合常理。

如下是被阻断域名（即在查询的过程中被劫持了，查询路径上某个服务器响应了一个伪造的 DNS 响应包，里面包含一个不正确的值，而这个包客户端无法分辨出真假）的 dig 例子。

```
$ dig www.example.com +trace
; <<>> DiG 9.8.3-P1 <<>> www.example.com +trace
;; global options: +cmd
.          3600 IN   NS   c.root-servers.net.
.          3600 IN   NS   b.root-servers.net.
.          3600 IN   NS   a.root-servers.net.
.          3600 IN   NS   m.root-servers.net.
.          3600 IN   NS   l.root-servers.net.
.          3600 IN   NS   k.root-servers.net.
.          3600 IN   NS   j.root-servers.net.
.          3600 IN   NS   i.root-servers.net.
.          3600 IN   NS   h.root-servers.net.
.          3600 IN   NS   g.root-servers.net.
.          3600 IN   NS   f.root-servers.net.
.          3600 IN   NS   e.root-servers.net.
.          3600 IN   NS   d.root-servers.net.
;; Received 505 bytes from 30.26.8.65#53(30.26.8.65) in 11 ms

www.example.com.    3335 IN   A    93.46.8.89
;; Received 66 bytes from 198.41.0.4#53(198.41.0.4) in 7 ms
```

缓存 DNS 劫持

如果配置的解析是到某 IP 地址 A，而在有的 Local DNS 上却响应的是地址 B，则可能是被缓存 DNS 劫持了。

国内的运营商有可能会在缓存 DNS 上做特殊的配置，将某些不存在的解析，或解析超时的解析指向其所谓的“纠错”页面，展示一些其运营商的业务内容。同时，如果运营商的缓存 DNS 安全性不足，则有可能被黑客利用，为缓存 DNS 投毒，导致域名被劫持到黑客设置的钓鱼网站。

如果发现域名在缓存 DNS 上解析出来的值不是自己设置的值，一定要提高警惕。

CNAME 值无法解析

因运维的方便，或因域名解析管理的授权，很多时候会将域名做 CNAME 记录，交由另一个域名的管理者来管理，以指向最终的 IP 地址。最典型的这类应用是 CDN 服务：网站管理者将自己的域名指向 CDN 厂商提供的域名，由 CDN 厂商再将域名指向各地的 CDN 节点 IP，或指向 NAME（或多级 CNAME）后，再指向最终的 IP。也有些虚拟网站提供商或其他互联网应用提供的是一个子域名，需要服务的使用者将域名添加 CNAME 类型的解析，以指向这个名字。

```
$ dig www.example.com +trace
; <<>> DiG 9.8.3-P1 <<>> www.example.com +trace
;; global options: +cmd
.          52904    IN   NS   g.root-servers.net.
.          52904    IN   NS   c.root-servers.net.
.          52904    IN   NS   e.root-servers.net.
.          52904    IN   NS   b.root-servers.net.
.          52904    IN   NS   k.root-servers.net.
.          52904    IN   NS   i.root-servers.net.
```

```
.          52904    IN   NS   j.root-servers.net.
.          52904    IN   NS   d.root-servers.net.
.          52904    IN   NS   m.root-servers.net.
.          52904    IN   NS   h.root-servers.net.
.          52904    IN   NS   f.root-servers.net.
.          52904    IN   NS   a.root-servers.net.
.          52904    IN   NS   l.root-servers.net.
;; Received 477 bytes from 30.26.8.65#53(30.26.8.65) in 40 ms

com.            172800   IN   NS   f.gtld-servers.net.
com.            172800   IN   NS   j.gtld-servers.net.
com.            172800   IN   NS   d.gtld-servers.net.
com.            172800   IN   NS   g.gtld-servers.net.
com.            172800   IN   NS   e.gtld-servers.net.
com.            172800   IN   NS   b.gtld-servers.net.
com.            172800   IN   NS   l.gtld-servers.net.
com.            172800   IN   NS   i.gtld-servers.net.
com.            172800   IN   NS   a.gtld-servers.net.
com.            172800   IN   NS   h.gtld-servers.net.
com.            172800   IN   NS   k.gtld-servers.net.
com.            172800   IN   NS   m.gtld-servers.net.
com.            172800   IN   NS   c.gtld-servers.net.
;; Received 492 bytes from 202.12.27.33#53(202.12.27.33) in 285 ms

example.com.        172800   IN   NS   ns1.example.com.
example.com.        172800   IN   NS   ns2.example.com.
;; Received 107 bytes from 192.52.178.30#53(192.52.178.30) in 121 ms

www.example.com.        600  IN   CNAME
www.example.com.bar.net.
;; Received 91 bytes from 1.1.1.1#53(1.1.1.1) in 288 ms
```

再去查 www.example.com.bar.net 时，却查不到对应的下一层记录了。

```
$ dig www.example.com.bar.net +trace

; <<>> DiG 9.8.3-P1 <<>> www.example.com.bar.net +trace
;; global options: +cmd
.          52820    IN   NS   c.root-servers.net.
.          52820    IN   NS   e.root-servers.net.
.          52820    IN   NS   b.root-servers.net.
.          52820    IN   NS   k.root-servers.net.
.          52820    IN   NS   i.root-servers.net.
.          52820    IN   NS   j.root-servers.net.
.          52820    IN   NS   d.root-servers.net.
.          52820    IN   NS   m.root-servers.net.
.          52820    IN   NS   h.root-servers.net.
.          52820    IN   NS   f.root-servers.net.
.          52820    IN   NS   a.root-servers.net.
.          52820    IN   NS   l.root-servers.net.
.          52820    IN   NS   g.root-servers.net.
;; Received 477 bytes from 30.26.8.65#53(30.26.8.65) in 40 ms

net.            172800   IN   NS   e.gtld-servers.net.
net.            172800   IN   NS   j.gtld-servers.net.
net.            172800   IN   NS   f.gtld-servers.net.
net.            172800   IN   NS   c.gtld-servers.net.
net.            172800   IN   NS   g.gtld-servers.net.
net.            172800   IN   NS   i.gtld-servers.net.
net.            172800   IN   NS   k.gtld-servers.net.
net.            172800   IN   NS   a.gtld-servers.net.
net.            172800   IN   NS   h.gtld-servers.net.
net.            172800   IN   NS   b.gtld-servers.net.
net.            172800   IN   NS   m.gtld-servers.net.
net.            172800   IN   NS   d.gtld-servers.net.
```

```
net.            172800   IN   NS   l.gtld-servers.net.
;; Received 510 bytes from 192.228.79.201#53(192.228.79.201) in 195 ms
bar.net.        172800   IN   NS   ns1.bar.net.
bar.net.        172800   IN   NS   ns2.bar.net
;; Received 213 bytes from 1.1.1.1#53(1.1.1.1) in 469 ms
bar.net.  300  IN   SOA  ns1.bar.net. hostmaster.bar.net. 20150101 3600
3600 360000 60
;; Received 121 bytes from 2.2.2.2#53(2.2.2.2) in 42 ms
```

部分解析异常

域名会指向两个或多个 DNS 服务器。但有时候会不小心同时指向多家 DNS 提供商的 DNS 服务器。如果服务器更换 IP 后只修改了其中一家 DNS 提供商的解析记录而忘记修改另一家了，就会造成解析有时候正常，有时候解析到旧的 IP 地址上，或者使用其中一家的 DNS 服务过期而清理了记录，也会造成解析不稳定，时好时不好，或部分地区不正常。

更换 DNS 但尚未完全生效

域名更换 DNS 提供商，或由第三方 DNS 提供商换为自己的 DNS 等，可能会涉及在新的 DNS 服务器上添加解析记录，然后在域名注册商处更改域名的 NS。但是，更改 NS 后不要急着在原 DNS 提供商处删除域名的记录。因更改 NS 后，最长 48 小时才能在全网生效，在这期间会有部分请求到原来的 NS 服务器上。如果原来的 NS 服务器上解析记录的整个域被清理了，则解析就会中断，造成部分用户无法访问。所以，更换域名 NS 时，一定要保证新老 NS 服务器上均保留一份正确的记录，且在老 NS 服务器上最少保留 48 小时。

resolv.conf 配置不当

默认情况下，解析域名超时时，会进行重试。但有时候为了提高性能，会把超时时间缩短或限制重试次数，而当 Local DNS 中无缓存时，解析的

时间会相对较长，也会导致解析偶尔失败，且问题不好处理，排查问题变得困难起来。如下面配置文件中的“options timeout:1 attempts:1 rotate”，即设置超时 1 时重试 1 次，这样的限制太严格了，建议删除这行配置而使用系统默认的值：

```
options timeout:1 attempts:1 rotate
nameserver ***
```

小结

域名是很多网络服务的入口，出现异常会造成很大损失，有时候由于各地缓存而造成域名问题无法快速恢复。

作为维护人员，在处理域名相关的变更时更需要多加注意。比如：域名是由权威 DNS 和各地运营商或第三方机构的缓存 DNS 等，在转移权威 DNS 服务提供商要在旧的 DNS 提供商中保留正确的解析至少 48 小时；在自己搭建 DNS 时，多加测试，注意兼容性；在配置多线路的智能解析（不同来源，指向不同的地址，即让 DNS 根据来源的不同而返回不同的值）时，要注意每个线路均能正常解析。出现问题时，可借助 dig/nslookup 等域名诊断工具以及域名的 whois 信息来诊断。

1.2 智能定位，网络端到端静默丢包点迅速锁定

背景

随着阿里巴巴集团业务的高速发展，数据中心的数量和规模都在快速扩张，网络运营的设备数量、型号、软件版本的类型均呈快速增长的趋势，

网络规模的扩张导致了相同故障发生概率下出现设备转发异常的次数也在不断增加，对业务影响和网络运营带来了很多风险和挑战。

常见的网络丢包故障发生在线路和端口级别，大都是因为线路质量劣化、线路流量拥塞、光模块故障等，这类问题网络运营团队有成熟的监控和自动化处理手段，可以确保故障被快速发现、定位和恢复。

在各类转发异常中，最困扰网络运营的难题就是静默丢包。静默丢包是发生在设备内部转发层面的，没有任何异常日志或告警的丢包故障。由于没有任何迹象可寻，静默丢包的发现和定位都是一个难题。

随着网络规模不断增加，网络设备静默丢包虽然为小概率事件，但其故障数量呈上升趋势，较长的故障定位和恢复时间都是业务所不能容忍的，网络运营是时候向静默丢包正式宣战了！

异常表现

静默丢包主要有下述异常表现。

（1）特定流持续不通。

（2）持续网络重传。

（3）特定大小报文丢包。

（4）固定时间出现丢包异常。

原因分析

若对静默丢包的原因进行详细分类，可以归为以下两类。

第一类：软失效

Parity Error 产生的原因是芯片软失效，Parity Error 是芯片软失效的一种校验方法。软失效是指高能粒子单元对芯片晶圆的撞击导致比特翻转而引起单比特错误、多比特错误及栓锁等，由于它对电路的损害不是永久性的，所以这种现象被称为软失效。

产生软失效的原因有以下三种。

（1）芯片封装材料含有放射性元素引发芯片软失效。

（2）来自宇宙和太阳系的高能中子芯片的原子核捕获产生附属的带电粒子，进而引发芯片软失效（是的，经常被大家调侃时说的太阳黑子爆发导致数据中心异常是真实发生的）。

（3）低能中子会和芯片内部绝缘体材料发生原子反应，产生高能粒子，引发芯片软失效。

软失效易发生于半导体存储芯片中（如 DRAM、SRAM、寄存器组），以及高速缓存和配置寄存器设备等。目前软失效无法完全避免，具有随机性和不确定性。软失效一般都是静默丢包问题的主要来源。

软失效又分 SBE（Single-Bit Error，单比特错误）和 MBE（Multi-Bit Error，多比特错误）两种情况，所谓单比特错误 SBE，是指在一个数据字中因为芯片软失效引起的一个比特的错误，而多比特错误 MBE 是指在一个数据字中因为芯片软失效引起的两个或多个比特的错误。通常，发生多比特错误的概率较低，但一旦发生，就极难纠正错误比特。

为了降低软失效问题发生的概率，芯片和半导体厂商分别从工艺和系统两个方面着手改进。工艺级措施方面选用放射性元素含量极低的封装材料和采用新的绝缘体材料技术，系统级措施采用奇偶校验（Parity

Protection）、纠错码（Error Correction Code，ECC）和双存储器的软硬件比较等技术来预防软失效的发生。

现有网络中各商用设备在关键转发表项上通过 ECC 字段来检测 SBE 和 MBE，当发生校验错误时通过发出告警信息告知用户，同时硬件可以通过 ECC 字段信息自动纠错 SBE，但无法纠正 MBE，MBE 则是通过上报中断，采用软复位（soft reset）、硬复位（hard reset）或软件重刷硬件表项。这些都依赖于芯片厂商的开发程度、检验和恢复机制，以及设备厂商的软件支持程度，如果芯片和 SDK 支持不足对一些非关键表项没有覆盖检测和自修复，又或者在芯片没有覆盖到的关键表项，设备厂家没有通过有效的软件机制解决都会导致网络设备出现静默丢包的问题。

第二类：设备硬件故障、软件缺陷和性能问题

设备单板芯片或外围器件硬件故障，软件本身机制考虑不全或实现有误等缺陷以及设备内部通道异常或拥塞都会导致业务出现丢包或异常篡改。对于设备硬件故障和软件缺陷之类的各个版本和厂家的问题具有多样性且无通用性，这里就不展开了。

故障发现

现有的静默丢包网络监控手段有传统手段（SNMP（简单网络管理协议）集采和实时系统日志分析）、主动测试发现手段（各种大小报文并且变换源目的 TCP 端口的 TCP fullmesh（全网状）ping）和人为可定义的巡检系统，但是受限于监控周期、巡检和 flow 覆盖能力以及安全要求，仍然会有一些特定五元组报文不通或间歇性丢包重传等静默丢包问题需要依靠业务上报。

故障定位和处理

由于业务提供访问异常流量的源目的在转发路径上会经过很多网络设备，在问题定位过程中需要在所有的网络设备端口上去部署流，然后根据比较计数来定位丢包点，这就要求对流统 flow（特定五元组的流量）灵活的打标和染色能力进行计数比较来确定问题。

这里分别从特定流不通和随机丢包两种情况来说明传统的定位方法和步骤。

第一类：特定流不通

这种问题属于静默丢包类最常见的问题，业务抓包可以发现特定五元组的包数据持续永久不通，对于这种转发路径上报文计数从有到无的异常，相对来说定位条件和难度较低。假设现在发现源端口 1234、目的 IP 和端口为 1.1.1.1:80 的 syn 包不通而源端口为 1235 时是通的，则可以使用命令：sudo traceroute -Tp 80 --sport=1234 1.1.1.1 和 sudo traceroute -Tp 80 --sport=1235 1.1.1.1，这就相当于分别用异常流和正常流去做 traceroute 比较看断点在哪里，在断点的上下游设备上分别做流通，这种方式也受限于服务器本身 traceroute 的版本和设备的 ttl 超时答复功能开启等，通用的方法是服务器构造持续发送不通的特定流，路径上做流统抓取从有到无的故障点。

第二类：随机丢包

这种问题对传统定位具有很大的挑战性，不是有和无的区别，而是多和少。正如前面描述的流统局限性，先要搭建一个和业务流相近的环境，然后在所有转发途径的设备端口部署流统，这需要大量的人力资源和时间成本。

综上所述，流统是一个原理简单、定位准确的静默丢包定位手段，但

是其最大的弊端是策略部署和报文统计环节的工作太重，需要在业务流所途径的所有 ECMP（Equal-CostMultipatchRoutig，等价路由）路径去部署策略并统计结果，人工实施一次流统大概耗时几个小时，如果一天要排查多个静默丢包的问题，其工作量可想而知。

由于这类问题的定位思路和方法是可以固化的，而且传统方式依赖于个人对设备命令的熟悉程度，并要自己去分析转发路径。因此，网络人员将固化的经验沉淀到系统中，以实现智能流统的自动化运营手段来提升整体工作效率。下面将详细描述静默丢包智能流统工具的实现和定位思路。

（1）寻找丢包五元组

由于业务人员的水平参差不齐，无法准确给出丢包的五元组，这就需要借助智能化流统工具来定位发生丢包的五元组。输入用户提供的源目的 IP 后，系统会以源端自动构造报文（源目的端口不断变化）发送到目的端，目的端进行报文分析并返回结果，如图 1-1 所示。

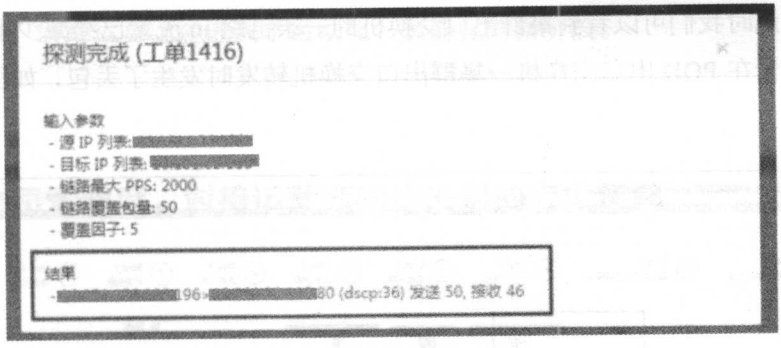


图 1-1　定位丢包五元组的结果

（2）自动发现网络拓扑、下发流统策略并发起测试

如图 1-2 所示，系统会根据输入的 AB 端服务器信息，自动发现途经的端到端网络拓扑，生成源到目的的转发路径，并下发流统策略，完成流

量策略部署后，系统会通过安装在服务器上的 AGENT，从服务器发起针对特定流的发包测试。

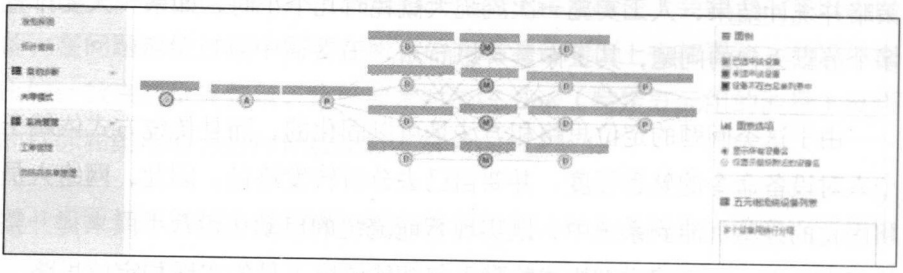


图 1-2　自动发现网络拓扑并部署流统策略

（3）流统结果采集和诊断结果的展示

这个步骤是将所有设备上的流统计数进行采集分析并图形化展示丢包位置，该步骤代替了人脑分析的过程，自动反馈有问题的故障点并标红，这个时候整个定位过程就算完成了。

此时我们可以看到集群出口交换机的一条链路 in 流量已经减少了，证明流量在 POD 出口交换机→集群出口交换机转发时发生了丢包，如图 1-3 所示。

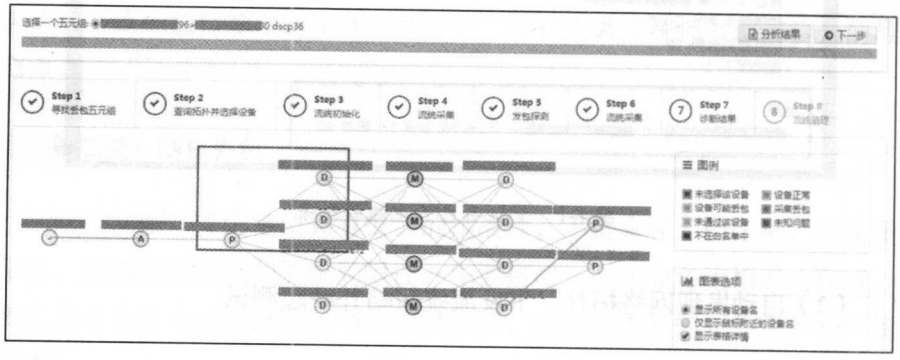


图 1-3　故障定位段落展示

在明确故障点之后，通常是优先恢复业务，由于网络设计上的冗余性，可以直接隔离异常芯片或板卡。通过隔离处理后发现业务恢复正常。

小结

通过智能流统的实现，逐渐将人从重复及固化的操作中解放出来。从人力上讲，传统方式需要多个跨团队的人员协作，演变成个人单兵作战完成问题定位；从时间上讲，由几小时简化到几分钟，大大降低了故障影响并缩短了定位时间。

未来，智能流统工具前向会和业务告警、网络监控系统结合，在业务异常时，自动启动并快速定位故障段落；后向会和网络自动化运维平台打通，在定位问题后触发自动化隔离，实现故障的自动发现、定位和解决，真正做到自动化运维。

随着更多的自动化运维工具的上线，也使网络运维自动化越来越迅捷和高效，网络运营已经进入了自动化、智能化的新阶段。

1.3　灵活调度，对接运营商网络流量的容灾策略

引子

客服人员："多个业务页面无法访问，集中在 A 地区，基本都为 X 运营商网路，已经收到大量客户反馈了。"

网络人员："A 地区 X 运营商网络问题，已联系运营商，已经在修复中了。"

客服人员："估计需要多久，用户们都很着急。"

网络人员："暂时无法确定，在等运营商回复。"

客服人员："恢复了吗？"

网络人员："估计还需要 2 小时。"

客服人员：……

这样的对话在客服人员和网络人员之间经常发生，也经常有人发出这样的疑问："当运营商网络出现问题时，用户真的只能干着急吗？"运营商修复时间一般较长，用户的业务受损是等不起的，那有什么其他办法降低影响吗？

背景

某日，网络运营团队通过互联网质量监控系统发现全国所有电信用户对华北一地区访问质量劣化，现象为该地区城网至电信全国所有方向网络发生大面积质量劣化，5 分钟后，网络人员快速执行流量调度容灾预案，将流量调度至其他地区后，用户业务影响恢复至正常水平。

为什么这次的运营商网络发生异常，网络运营团队可以快速恢复呢？下面会详细说明运营商容灾切换能力的建设过程。

原因分析

阿里巴巴在国内多个区域对外提供服务，每个区域都和国内的三大运营商及重要的二级运营商建立了带宽充足的互联电路。

如果在阿里巴巴某区域当地互联的运营商网络内发生故障，就会导致

该运营商的所有用户访问阿里巴巴该区域的业务时，发生访问质量劣化，也就是本文开始时发生的那一幕。如果运营商骨干网发生故障，则可能导致运营商网内用户访问阿里巴巴多个区域时均发生不同程度的质量劣化。在运营商网内发生上述故障时，通过几个层面的合理流量调度，可以消除故障影响，快速恢复业务。

业务自身调度

某用户如果购买了多个区域的阿里云资源向外提供服务，并且实现了较好的业务容灾机制，当区域 A 的运营商 X 网络发生故障时，用户可以通过 DNS 容灾切换或 CDN 调度容灾切换，快速将区域 A 的业务切换到区域 B，此时运营商 X 全国的用户访问原来受影响的区域 A 的服务时，实际访问路径已经切换到了区域 B，成功绕开了运营商 X 在区域 A 的网络故障点，快速恢复了业务。

对于重要业务，特别是易受到运营商网络内的时延大、抖动、丢包影响的网络质量敏感型业务，应重点建设业务层面的多地容灾切换机制，确保业务快速恢复。

阿里巴巴网络的区域出口容灾调度

要实现网络级别的快速容灾调度，秒级的互联网监控发现能力、故障调度决策系统和骨干网级别的 SDN 调度能力都是必不可少的。

秒级互联网监控系统：通过在运营商网内部署监测点，实时发起流量监控，阿里巴巴的业务监控团队可以模拟运营商用户对阿里巴巴业务的访问请求，一旦运营商网络内发生故障，该监控能力可以实现秒级的故障发现和定位，故障定位的精细程度可以细化到运营商网内地市级故障、省网级故障、骨干网级故障及阿里巴巴区域出口级故障等，应该说故障发生的位置距离阿里巴巴某区域和运营商互联的出口越近，则对阿里巴巴该区域

的业务影响也就越大，当然运营商骨干网级别的故障可能对阿里巴巴多个区域的业务都会造成影响。

容灾调度决策系统：秒级互联网监控系统会将发现和定位的异常告警发送给容灾调度决策系统，决策系统会根据问题严重程度和故障定位情况，决策是否执行容灾切换，并将决策指令发送给阿里巴巴骨干网 SDN 调度系统。

阿里巴巴骨干网 SDN 调度系统：收到决策系统的调度指令后，SDN 调度系统会通过调整阿里巴巴网内路由策略及和运营商互联的网间路由策略，实现运营商出口容灾切换，将区域 A 和运营商 X 的全部互联流量，双向都调整到区域 B 和运营商 X 互联的电路带宽上；业务访问的路径将从运营商直接访问区域 A，变为运营商访问区域 B，流量到达区域 B 后再通过阿里巴巴内网到达区域 A，从而绕开运营商网内的故障点。

运营商出口容灾切换相比第一点提到的业务自身的容灾切换，其影响面较大，被切换区域的所有业务访问路径都会发生变化，伴随而来的是访问时延可能会造成一定增加，一般只有在区域 A 直连的运营商网络发生重大故障，影响了该区域所有业务访问时，决策系统才会决策这种区域出口级别的切换。

阿里巴巴网络的精细化容灾调度

如果运营商骨干网络发生故障，则现象是运营商部分省访问阿里巴巴区域 A 的业务质量发生劣化，此时如果决策区域出口切换动作显得太重，而且不能有效解决问题（即使将流量调整到区域 B，运营商网内仍有部分省访问区域 B 不好），就需要更加精细的 SDN 容灾调度能力。

对于这种 SDN 调度能力，阿里巴巴的出流量可以实现 100%的精准调度，入流量的调度还要依靠运营商网络架构的合理性和规范性来配合实现。

小结

综上所述，经过阿里巴巴网络运营团队长期的经验沉淀、总结提炼以及工具开发，通过互联网秒级监控系统、容灾决策系统和骨干网 SDN 调度系统的实现，对于运营商网内的故障，再也不是无计可施，而是拥有丰富的容灾切换能力，进一步结合业务自身的容灾切换能力，可以实现大部分问题的快速解决。

1.4　抽丝剥茧，深挖云盘挂起背后的真相

引子

阿里云使用飞天分布式存储系统作为虚拟机云盘的终端设备，其主要有三个角色。

- master：负责集群元数据的管理、集群健康检查、数据安全等，多 master 之间采用 primary-secondaries 模式，基于 PAXOS 协议来保障服务高可用，primary 在地址服务器注册访问地址，多 master 在检测到 primary 异常后会自动修复，选举出新的 primary。
- chunkserver：负责实际数据存储，通过冗余副本提供数据安全性。
- client：对外提供类 POSIX 的专有编程接口，存储服务的入口，满足多种场景下数据存储的需求。通过查询地址服务器解析 primary master 地址，从 primary 获取元数据信息后和 chunkserver 交互完成数据存储。

背景

某日凌晨 2 点 45 分，阿里云 ECS 某测试集群存储系统 primary master 发生切换，随后陆续有虚拟机报警发生云盘挂起。2 点 53 分，集群异常虚拟机开始逐渐恢复，2 点 54 分，除个别长尾外，全部恢复正常。后来，开发人员调查发现该集群存储系统原 primary master 发生了磁盘故障，并在 13 秒后主动触发了切换，新 primary 状态正常。磁盘故障后，master 行为完全符合预期，理论上虚拟机应该无感知，但现实是仍有部分虚拟机发生了云盘挂起。

原因定位

存储系统 primary master 因磁盘阵列卡或磁盘故障造成不可服务的问题之前也遇到过，为此，技术人员开发了存储健康检查功能，当 primary master 发现服务异常后会触发 primary 自动切换。通过分析新 primary master 日志，确认此功能正常，老 primary 磁盘故障发生 13 秒后就主动触发并完成重新选举，新 primary 开始接受请求并对外提供服务。

存储系统新 primary 选举出来后，仍有虚拟机发生云盘挂起的现象和预期完全不符。因为老 primary 磁盘故障后先转变成只读，之后服务器重启以恢复服务，造成老 primary 服务日志全部丢失，这增加了调查的难度。后期通过仔细调查分析，发现如下不可解释的现象。

- 新 primary 服务器有老 primary 主动触发切换相关的日志。
- 老 master 在切换后直接调用地址服务接口停止 primary 地址更新，同时调用 ServiceExit 退出，全路径无磁盘操作，理论不受磁盘影响。
- chunkserver 上和老 master 相关的远程调用（RPC）调用在 2 点 48 分停止。
- 发生云盘挂起的虚拟存储系统 client 持续访问老的 master。

因为当前运行的存储系统软件支持 Primary Master 地址缓存功能，所以最初怀疑异常的虚拟机是否和缓存没有及时更新有关，但经过进一步排查，排除了这种可能，存储系统新老 primary 从 02:45:17 到 02:52:55 在地址服务器频繁交替更新 primary master 的服务地址。

```
time:  20xx-xx-xx  02:45:17,691  session:0x1e56c66df7367b06  cxid:0x1e
zxid:0xb0049d669  opeatetype:create  path:/cluster/sys/service/master  ephemeral:True
acl:[{perm:cdrwa, id:{schema:world, id:anyone}}]  data:tcp://a.b.c.d:10250
time:  20xx-xx-xx  02:45:17,882  session:0x1e56c66df7367afd  cxid:0x3d
zxid:0xb0049d675  opeatetype:create  path:/cluster/sys/service/master  ephemeral:True
acl:[{perm:cdrwa, id:{schema:world, id:anyone}}]  data:tcp://a.b.c.e:10250
.....
time:  20xx-xx-xx  02:52:54,903  session:0x1e56c66df7367afd  cxid:0x3592
zxid:0xb004a4d6a  opeatetype:create  path:/cluster/sys/service/master  ephemeral:True
acl:[{perm:cdrwa, id:{schema:world, id:anyone}}]  data:tcp://a.b.c.d:10250
time:  20xx-xx-xx  02:52:55,030  session:0x1e56c66df7367b06  cxid:0x31c8
zxid:0xb004a4d72  opeatetype:create  path:/cluster/sys/service/master  ephemeral:True
acl:[{perm:cdrwa, id:{schema:world, id:anyone}}]  data:tcp://a.b.c.e:10250
```

由上述现象可知，老 primary 进程的服务线程都已经停止工作，但是负责更新 primary master 服务地址的线程没有退出，并持续更新。在长达 7 分钟的时间内，更新线程能正常工作，无任何磁盘操作的 ServiceExit 未成功，后台负责停止更新线程服务的线程也不能正常执行。

根据存储系统的设计，primary master 在启动后就会有一个单独的线程定期检查一个 volatile 变量 mMasterHasSuicide，如果 mMasterHasSuicide 为 true，直接调用 AddressResolver::Unregister 停止更新服务地址。

mMasterHasSuicide 设置为 true 是在触发切换后第一条指令，切换完成意味着 mMasterHasSuicide 肯定已经为 true。另外，更新完 mMasterHasSuicide 后，老的 primary master 也会走自杀流程。

技术人员最初怀疑日志操作可能因磁盘故障造成退出失败，但经过仔细检查复查代码确认 master 使用全异步日志模式，服务线程只在内存中生成日志内容，由后台写盘线程负责日志落盘，服务线程不会因为磁盘故障阻塞。

存储团队的开发尝试多种方法调查问题的根本原因，但都收获不大。最终在翻查系统日志时，无意发现/var/log/check_hw/basemessages 记录了大量 IO 异常的日志，时间点恰好是问题恢复后。

```
X 月 X 日  02:52:52 localhost: : megasas: [ 0]waiting for 211 commands to complete
X 月 X 日  02:52:52 localhost: : megasas: [ 5]waiting for 211 commands to complete
......
X 月 X 日  02:52:55 localhost: : megasas: [165]waiting for 211 commands to complete
X 月 X 日  02:52:55 localhost: : megasas: [170]waiting for 211 commands to complete
......
X 月 X 日  02:52:55 localhost: : sd 0:2:1:0: timing out command, waited 360s
X 月 X 日  02:52:55 localhost: : sd 0:2:1:0: SCSI error: return code = 0x06000000
X 月 X 日  02:52:55 localhost: : end_request: I/O error, dev sdb, sector 101492738
X 月 X 日  02:52:55 localhost: : Buffer I/O error on device sdb1, logical block 12686588
X 月 X 日  02:52:55 localhost: : lost page write due to I/O error on sdb1
......
X 月 X 日  02:52:55 localhost: : sd 0:2:0:0: SCSI error: return code = 0x06000000
X 月 X 日  02:52:55 localhost: : end_request: I/O error, dev sda, sector 857868853
X 月 X 日  02:52:55 localhost: : sd 0:2:0:0: timing out command, waited 360s
......
X 月 X 日  02:52:55 localhost: : sd 0:2:0:0: timing out command, waited 360s
X 月 X 日  02:52:55 localhost: : sd 0:2:0:0: SCSI error: return code = 0x06000000
X 月 X 日  02:52:55 localhost: : end_request: I/O error, dev sda, sector 879221669
X 月 X 日  02:52:55 localhost: : Aborting journal on device sda6.
X 月 X 日  02:52:55 localhost: : Remounting filesystem read-only
```

根据统计结果，从 02:52:52 到 02:52:55 总共有 211 个扇区发生磁盘操作超时，通过把日志中的扇区转换成文件系统块号，然后使用系统工具

debugfs 的 icheck 命令可以查询其隶属的 inode，使用 ncheck 命令可以完成 inode 到文件名的转换。但是在实际使用中发现，查询过程很慢，期间 utility 磁盘利用率一直保持在 100%。既然通过文件系统块号查文件名方案不行，那反过来查是不是就没那么慢了呢？通过如下命令可以快速查询某个磁盘设备（比如挂载到/apsara 的 sda6）上文件和块号的对应关系。

```
[admin@master]$ find /apsara -type f |awk '{print "hash " $0 "\n" "stat " $0}'|sed 's|/apsara||'|sudo debugfs
/dev/sda6
```

得到输出结果之后，技术人员写了一个脚本，解析输出结果构建块号到文件名的索引表，并扫描了 211 个扇区对应的文件。经过分析发现，大部分报告磁盘操作超时的扇区对应文件已经删除，没有删除文件也基本是日志文件，在过滤掉这些文件后，剩余两个程序的可执行文件。

```
storage_master
    \--- block:69757970 sector:857868853
ping_agent
    \--- block:72427072 sector:879221669
```

根据文件名和块号，可以获取块对应的 storage_master 文件内容。

```
debugfs:  stat storage_master
Inode: 34865170   Type: regular    Mode:  0775   Flags: 0x0   Generation: 2476818772
User:     0   Group:     0   Size: 313103822
File ACL: 0    Directory ACL: 0
Links: 2   Blockcount: 612144
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x57e47a6f -- 08:42:23 2016
atime: 0x582a9814 -- 13:07:32 2016
mtime: 0x57e47a11 -- 08:40:49 2016
BLOCKS:
(0-11):69756933-69756944,  (IND):69756945,  (12-1035):69756946-69757969,  (DIND):69757970,
(IND):69757971, (1036-2059):69757972-69758995, (IND):6
9758996, (2060-3083):69758997-69760020, (IND):69760021, (3084-4107):69760022-69761045
```

```
.....
TOTAL: 76518
```

从上面的信息可知，块 69757970 正好对应 master 文件在 ext3 文件系统中的二级索引表（DIND）。所以，理论上，这个块发生读超时的话，超时期间，master 从第 1036 个逻辑块开始的数据都不能访问了。基于上面的日志信息，技术人员推测，对于 master 的代码，操作系统默认不会把所有数据都从磁盘加载到内存中，只有在执行过程中，访问内存发生缺页中断，master 才会把相应的代码从磁盘加载到内存并继续执行，而如果发生磁盘挂起的情况下，读 master 程序主体数据在访问二级索引随之挂起，导致需要的代码不能加载，也就不能执行，从而触发了虚拟机的云盘挂起。

细节分析

为了证明这个问题，需要确认 AddressResolver::Unregister 和 ServiceExit 的代码在问题期间没在内存中，但磁盘故障机器已经被重启，不能直接验证这个猜测，所以技术人员用新 primary master 内存状态间接验证这个结论。

```
[admin@master]$ readelf -e storage_master|egrep "VirtAddr|MemSiz|LOAD"
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD      0x0000000000000000 0x0000000000400000 0x0000000000400000

[admin@master]$ echo _ZN6apsara15AddressResolver10UnregisterERKSs |c++filt
apsara::AddressResolver::Unregister(std::basic_string<char, std::char_traits<char>,
                                                    std::allocator<char> > const&)

[admin@master]$ objdump -T storage_master|grep _ZN6apsara15AddressResolver10UnregisterERKSs
00000000025fc5a0 g     DF .text  0000000000000192  Base
_ZN6apsara15AddressResolver10UnregisterERKSs
```

```
[admin@master]$ a=$(((0x25fc5a0 - 0x400000) / 4096));b=$(((0x25fc5a0 + 0x192 - 0x400000) / 4096))
[admin@master]$ printf "%lx,%lx %lu,%lu" $a $b $a $b
21fc,21fc 8700,8700

[admin@master]$ echo _ZN6apsara13StorageMaster11ServiceExitEi |c++filt
apsara::StorageMaster::ServiceExit(int)

[admin@master]$ objdump -T storage_master|grep _ZN6apsara13StorageMaster11ServiceExitEi
0000000002125960 g     DF .text  00000000000007cf  Base
_ZN6apsara13StorageMaster11ServiceExitEi

[admin@master]$ a=$(((0x2125960 - 0x400000) / 4096));b=$(((0x2125960 + 0x7cf - 0x400000) / 4096))
[admin@master]$ printf "%lx,%lx %lu,%lu" $a $b $a $b
1d25,1d26 7461,7462

[admin@master]$ sudo ./vmtouch storage_master -v
                  0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
0x00001d00    11010000111111111111111111111111111111110000011111000000010111100
[43 vs 21]
0x000021c0    00000000000000000001111111110101111111111111111101110000000000001
[30 vs 34]
```

通过上面的分析信息可知，storage_master 物理文件从位移 0 开始的内容被映射到从 0x400000 地址开始的一段内存空间。所以 AddressResolver::Unregister 和 ServiceExit 分别对应文件的逻辑块号[8700, 8700]和 [7461, 7462]都需要 storage_master 文件的二级索引信息才能正常访问。

Linux 提供的 mincore 函数能够判断一个文件的某些块是否在内存中，在 primary master 正常运行一段时间后，我们使用基于 mincore 的开源工具 vmtouch 查看代码在内存中的加载情况：

```
[admin@master]$ sudo ./vmtouch storage_master -v
                  0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
```

```
0x00001d00    11010000111111111111111111111111111111110000011111000000000010111100 [43 vs 21]
0x000021c0    00000000000000000000111111111010111111111111111110111000000000000001 [30 vs 34]
```

分析发现，块 7461(0x1d25)在内存中，块 7461(0x1d26)和 8700(0x21fc)不在内存中，块 7461 因为 ServiceExit 前面相邻代码被执行过，所以被捎带加载了，而 ServiceExit 核心代码及地址服务的 Unregister 相关代码则因为从来没有被执行过而没有机会被加载，符合之前的推测。

确定原因后，我们模拟触发场景做了故障重现。

- 把存储服务 storage_master 程序主体在内存中的缓存清除。
- 启动前端压力，让服务稳定运行一段时间。
- 加载磁盘读写劫持内核模块，drop 掉 master binary 所在磁盘的所有 IO。

分析模拟场景下存储服务 master 行为确定和异常集群相同，master 没有退出，地址服务线程仍继续更新地址，最终确定问题原因。

解决方案

短期方案

- 在存储系统 master 进程启动后，master 会通过 mlock 方法把程序主体对应的数据加载并锁定到内存中。
- 地址服务不再通过调用方法停止 primary 地址更新，而是在地址更新主路径检测某个变量的值，判断是否需要继续更新。

长期方案

- 存储系统去除对地址服务的强依赖，存储系统的客户端启动后会缓存 master 地址信息，并内部维护 master 地址的正确性，不再强依赖外部地址解析服务。

- 地址服务提供基于 version 的地址更新接口，避免老 master 用过期地址覆盖有效地址。

小结

存储服务 master 模块为了应对类似的磁盘故障，在开发过程中被精心设计了双重保险以确保其退出或者无害，其一是调用 ServiceExit(内部调用 _exit)以强制跳楼的方式退出；其二是调用地址服务的反注册接口，要求其不再更新老 master 地址，这样即使没有及时退出也无害，并且确保这两条路径中都没有阻塞的同步磁盘操作，以避免磁盘故障导致线程挂起。

本次磁盘故障中，双重保险的两个独立线程异常处理流程从来没有被跑到过，所以运行到相关路径时发现对应的代码段都不在内存中（线下演练，发现这是一个概率事件，与编译环境、运行环境、内存状况等众多基础环境因素相关），运行时产生缺页中断，中断处理程序会从磁盘上加载对应的代码段，因为磁盘阵列卡或磁盘故障，导致内核的中断处理线程挂起，代码段加载不回来，进而导致用户态的这 2 个线程也挂起了；此时另外一个地址服务后台更新线程因为从启动开始就更新 master 地址，所以一直在内存中，从而运行没有受影响，在新 master 向地址服务器设置好正确地址后，它会反复用已经过期的老地址覆盖正确值，导致客户端解析出来的存储系统 master 地址一直在新老地址之间变化，最终部分虚拟机因为每次都拿到错误地址，造成云盘挂起。

1.5 存储的底线，SSD 数据不一致

背景

某业务方开发人员发现服务器异常掉电后，系统的文件数据不完整。

如图 1-4 所示，开发人员认为在 035C7000 这个文件地址之后，仍然有 30KB 的数据，但服务器重启之后，文件显示这部分为空，直接读取数据发现全为 0。他们怀疑这台机器的存储硬件（PCIE-SSD）出现了掉电丢数据的问题。

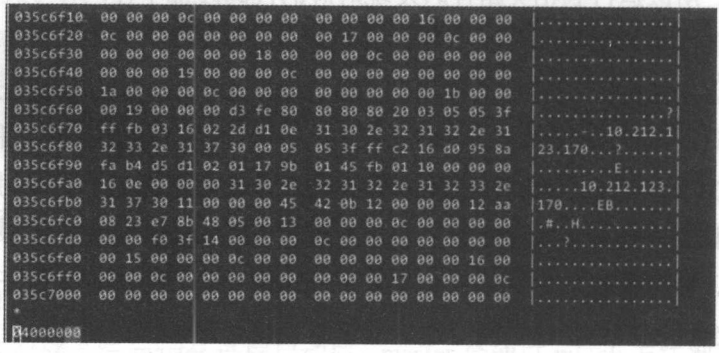


图 1-4 系统文件数据异常日志

问题排查

接到开发人员的问题，存储团队立刻进行了跟进，首先分析了 PCIE-SSD 的掉电数据保护机制。

PCIE-SSD 对于异常掉电，有完善的保护机制，且根据之前的白盒黑盒的数据一致性测试和反复掉电测试，未发现问题，初步排除硬件方面的问题。

深入分析

通过和开发人员的沟通，存储团队了解到业务方使用 pwrite 写数据，使用 O_DIRECT，没有加 O_SYNC。

这是一个非常大的怀疑点，如果没有 O_SYNC，文件系统的元数据可能在掉电时并未写入 PCIE-SSD（O_SYNC，在每个写操作，会等待磁盘返还结果后才返回，确保了数据一定落盘）。

但是要验证这个猜想，需要证明数据在 PCIE-SSD 上，仅是文件系统读不到这个数据（因为没有文件系统元数据）。

如图 1-5 所示，图中的 0000512 这一行，就是上面的 035C7000。继续往下读（如图 1-6 所示）。

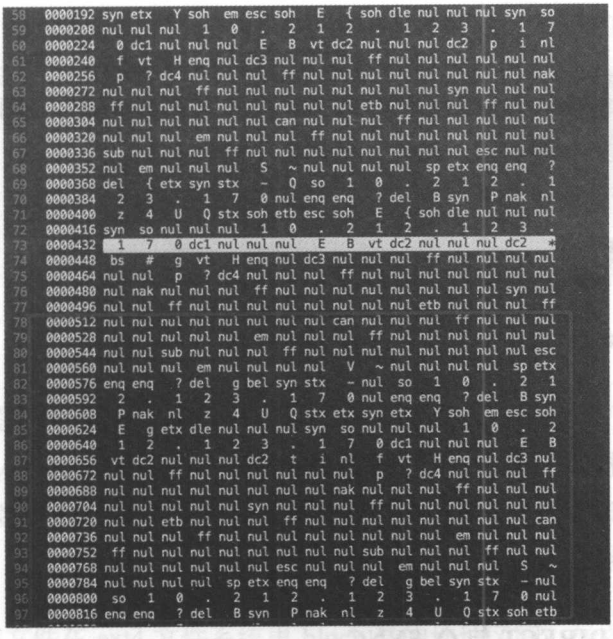


图 1-5　直接读取 Aliflash 裸数据

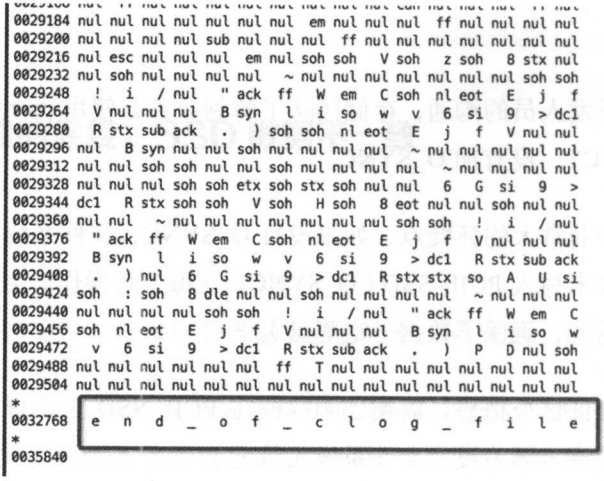


图 1-6 读取文件

连文件末尾的结束符都对的上，大小还刚好是业务方丢的 30KB。将此发给业务方确认，数据的确在盘上。

由于文件系统元数据未刷入磁盘，导致文件系统层面没有这个映射关系，所以读文件读不出，造成了数据丢失的假象。

相关知识点

（1）ext4 元数据

元数据（metadata）是文件系统用于描述用户数据的数据。它包含文件时间戳、权限、block 地址映射表等内容。如果用户数据写入文件时，元数据没有及时更新，同样会导致文件数据丢失。因为 block 映射表未更新，文件不知道哪些 block 是其组成部分。

（2）O_DIRECT 与 O_SYNC

用户使用 pwrite 写数据，加上 O_DIRECT 标识，只能保证数据直接落

盘（忽略 Buffer cache），而文件系统元数据仍然存储在 inode cache（内存）中，异常断电时，即使存储设备可以做到掉电保护（Power loss protection，企业级 SSD 特性），但服务器内存里的 inode cache 还未来得及写入存储设备，会造成元数据丢失。

当加上 O_SYNC 标识，写操作变为同步写（synchronous I/O），此时可以保证元数据同步落盘。见 man 手册中的说明：The O_DIRECT flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the O_SYNC flag that data and necessary metadata are transferred. To guarantee synchronous I/O, O_SYNC must be used in addition to O_DIRECT.

（3）文件在存储设备上的实际位置（LBA）

在本次的问题分析中，通过文件名找到了数据存储在 PCIE-SSD 的实际位置（LBA），读出了 PCIE-SSD 的裸数据，下面以/var/log/dmesg 文件举例，操作命令如下：

找到该文件所在的磁盘分区，如图 1-7 所示，该文件属于 sda3。

```
#df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        50G  8.0G   39G  18% /
devtmpfs        126G     0  126G   0% /dev
tmpfs           126G  268K  126G   1% /dev/shm
tmpfs           126G   61M  126G   1% /run
tmpfs           126G     0  126G   0% /sys/fs/cgroup
/dev/sda2       976M  130M  780M  15% /boot
/dev/sda6       110G  2.1G  108G   2% /apsara
/dev/sda5        55G   53M   55G   1% /apsarapangu
tmpfs            26G     0   26G   0% /run/user/0
```

图 1-7　使用 df 获取分区和目录信息

找到该文件在 ext4 文件系统里 block 号的地址区间，如图 1-8 所示，属于 block 4235796~4235820。注意这里的单位是 block，默认 ext4 的 block

是 4KB，即 8 个扇区（512byte）。

```
#filefrag -v /var/log/dmesg
Filesystem type is: ef53
File size of /var/log/dmesg is 100351 (25 blocks of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..      24:    4235796..   4235820:     25:             eof
/var/log/dmesg: 1 extent found
```

图 1-8　使用 filefrag 获取 block 地址区间

找到分区/dev/sda3 的起始 LBA 地址。如图 1-9 所示：2105344，注意这里的单位是扇区（512byte）。

```
#parted /dev/sda
GNU Parted 3.1
Using /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) unit s
(parted) p
Model: ATA INTEL SSDSC2BB24 (scsi)
Disk /dev/sda: 468862128s
Sector size (logical/physical): 512B/4096B
Partition Table: gpt
Disk Flags:

Number  Start       End         Size        File system     Name     Flags
 1      2048s       8191s       6144s                               bios_grub
 2      8192s       2105343s    2097152s    ext4                    boot
 3      2105344s    106962943s  104857600s  ext4
 4      106962944s  111157247s  4194304s    linux-swap(v1)
 5      113281024s  230469631s  117188608s  ext4            logical
 6      230469632s  464842751s  234373120s  ext4            logical
```

图 1-9　使用 parted 获取分区起始 LBA 地址

文件 LBA 地址 =（block 号 × 8）+ /dev/sda3 的起始 LBA 。

dmesg 文件 LBA 地址 = 4235796 × 8 + 2105344 = 35991712。

读出裸数据，注意要将 LBA 转换为 byte，乘以 512 即可。35991712 * 512 = 18427756544。如图 1-10、1-11 所示，直接读文件和读裸设备，输出一样，验证成功。

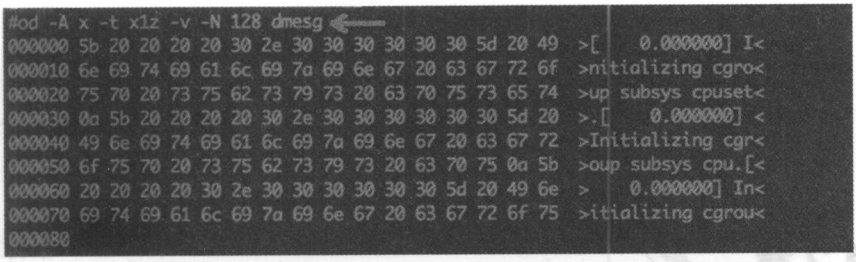


图 1-10　使用 od 读取文件头部

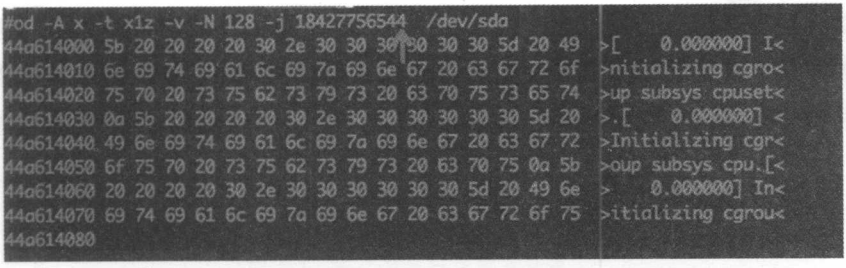


图 1-11　使用 od 读取裸设备偏移量

小结

本文这个案例虽然只是虚惊一场，但对于业务方提出的丢数据排查，不仅需要理解业务的写入模式，还需要名下 DirectIO，Sync 的机制和文件系统的刷盘原理。

整个 IO 链路理解的越清楚，就越容易看到问题背后的真相。

第2章 中间件使用常见隐患与预防

从 2007 到 2008 两年间，淘宝架构从集中式架构快速演变为分布式架构，并在业务上分层，把公共业务逻辑抽象成服务中心，提供一种服务能力，通过分布式中间件进行远程消息传递、服务调用，通过软负载实现流量的负载均衡，底层有存储集群。中间件技术作为阿里巴巴生态系统的技术基石，为全集团提供着可靠、高效、易扩展的技术基础服务。

洪峰的流量和多样性的使用场景，给中间件稳定性保障带来了极大的挑战，在技术发展的历史中，也遇到了很多有代表性的问题，本章节就会从用户和产品视角来介绍几个经典的案例，剖析原因，提出解决方案，希望对大家有所帮助，避免类似的问题发生。

2.1　高并发“热点”缓存数据快速“退火”

背景

电商场景促销活动的会场页由于经常集中在某个时间点进行“秒杀”促销，这些页面的 QPS（服务器每秒可以处理的请求量）往往特别高，数据库通常无法直接支撑如此高 QPS 的请求，常见的解决方案是让大部分相同信息的请求都尽可能地压在缓存（cache）上来缓解数据库（DB）的压力，从而尽可能地去满足高并发访问的诉求（如图 2-1 所示）。

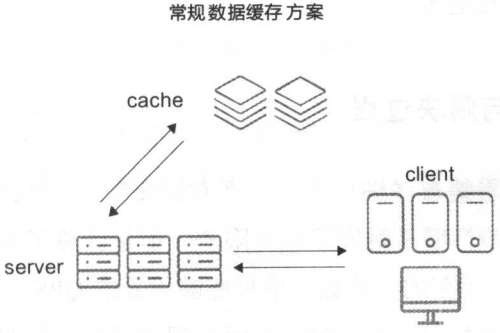


图 2-1　常规数据缓存方案

在一次业务促销过程中，运营给一大批用户集中推送了一条消息：10 点钟准时抢购一批远低于市场价而且数量有限的促销活动商品。由于确实物美价廉，用户收到消息之后 10 点钟准时进入手机客户端的会场页进行疯抢。几分钟内很多用户进入会场页，最终导致页面异常，服务器疯狂报警。报警信息显示很多关于缓存的异常，由于缓存拿不到数据转而会转向数据库去查询数据，这样数据库更加难以支撑，整个业务集群处于雪崩状态（如图 2-2 所示）。

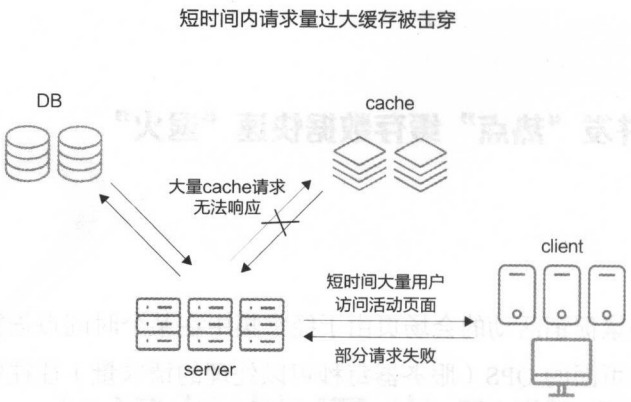


图 2-2 短时间内请求量过大缓存被击穿

此时缓存到底发生了什么问题？关注哪些方面可以有效地预防缓存被击穿导致雪崩的发生呢？

缓存问题分析与解决过程

（1）首先查看缓存详细日志，发现有很多带有“CacheOverflow”字样的日志，初步怀疑是触发了缓存限流。但是计算了缓存的整体能力和当前访问量情况：缓存的机器数×单机能够承受的 QPS > 当前用户访问的最大 QPS 值，此时用户访问 QPS 并没有超过缓存之前的预算，怎么也会触发限流呢？

（2）进一步分析日志，发现所有服务器上限流日志中缓存机器 IP 貌似都是同一台，说明大流量并没有按预想平均分散在不同的缓存机器上。回想前面提到的案例实际现象，发现确实有部分数据用户的访问请求都会触发对缓存中同一个 key（热点 key）进行访问，用户访问 QPS 有多大，则这个 key 的并发数就会有多大，而其他缓存机器完全没有分担任何请求压力，如图 2-3 所示。

（3）然后紧急梳理出存在“热点请求”的 key，并快速接入“热点本

地缓存”方案，然后迅速在下一场秒杀活动中进一步进行验证，此时发现之前异常大幅度减少。不过还是有少量“CacheOverflow”字样异常日志。热点 key 的请求都被“本地缓存”拦截掉了，此时发现远程 QPS 限流异常已经基本没有了，这又是什么原因呢?

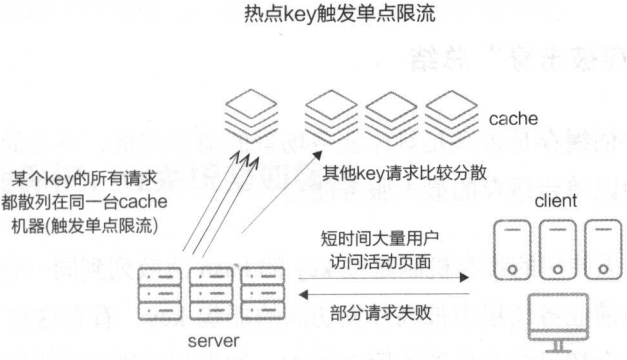


图 2-3　热点 key 触发单点限流

仔细查看缓存单台机器的网络流量监控，发现偶尔有网络流量过大超过单台缓存机器的情况（如图 2-4 所示）。

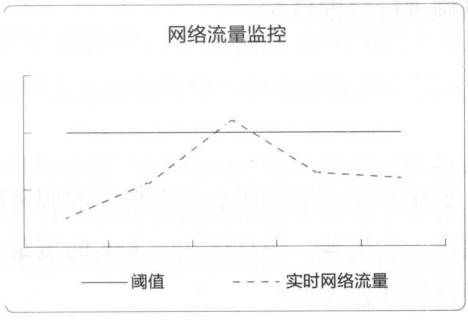


图 2-4　网络流量监控

说明缓存中有某些 key 对应的 value 数据过大，导致尽管 QPS 不是很高，但是网络流量（QPS × 单个 value 的大小）还是过大，触发了缓存单台机器的网络流量限流。

（4）紧急梳理出存在“大 value”的 key，发现这些“大 value”部分是可以精简，部分是可以直接放入内存不用每次都远程获取的，经过一番梳理和优化之后，下次“秒杀”场景终于风平浪静了。至此问题初步得到解决。

预防“缓存被击穿”总结

（1）评估缓存是否满足具体业务场景的请求流量，不是简单地对预估访问流量除以单台缓存的最大服务能力。

（2）如果使用的缓存机制是按 key 的 hash 值散列到同一台机器，则必须梳理出当前业务场景中被高并发访问的那些 key，看看这些 key 的并发访问量是否会超过单台机器的服务能力，如果超过则必须采取更多措施进行规避。

（3）除了关注 key 的并发访问量外，还要关注 key 对应 value 的大小，如果 key 的并发访问量 × value 大小 > 单台缓存机器的网络流量限制，则也需要采取更多措施进行数据精简。

更多思考

（1）单个 key 的请求量不超过单台缓存机器的服务能力，但是如果多个 key 正好散列到同一台机器，而且这几个 key 的流量之和超过单台机器的服务能力，我们该如何处理呢？

（2）单个 key 的并发访问量 × 对应 value 大小 < 单台缓存机器的网络流量限制，但是如果多个 key 的并发访问量 × 各自对应 value 大小 >单台缓存机器的网络流量限制，又该如何处理呢？

针对上述两个问题，首先要做的是做好缓存中元素 key 的访问监控，

一旦发现缓存有 QPS 限流或者网络大小限流时，能够迅速定位哪些 key 并发访问量过大，或者哪些 key 返回的 value 大小较大，再结合缓存的散列算法，通过一定规则动态修改 key 值来自动将这些可疑的 key 平均散列到各台缓存机器上去，这样就可以充分地利用所有缓存机器来分摊压力，保证缓存集群的最大可用能力，从而减少缓存被击穿的风险。

2.2　自我保护，让系统坚如磐石

背景

对一个应用开发者来说，希望自己的应用访问量越大越好。但是另一方面，因为应用的机器数量和配置有限，不可能承担任意多的流量。为了保证正常的服务，应用必须能够在访问量超过本身服务能力时拒绝一部分请求，保证不被打垮。当然，限流保护必须和一整套的容灾体系配合使用才能起到自我保护的作用，否则，非但达不到自我保护的目的，还有可能会造成调用方异常甚至资损。本文将结合阿里巴巴的一个软负载系统 VIPServer 从实际案例出发讨论在生产中限流保护应该如何实施。

首先，本文将简单介绍 VIPServer 整体的容灾体系；其次，结合实际案例说明没有做好自我保护可能造成的重大损失；再次，介绍 VIPServer 是如何把自己的容灾体系和限流保护结合，确保自身安全的同时又能保证业务应用的正常运行。

VIPServer 容灾体系

VIPServer 是阿里巴巴非常重要的一款软负载产品，负责了阿里巴巴几乎所有 http 请求的流量负载均衡，为阿里巴巴的异地多活和同城容灾提供

了非常重要的支撑。VIPServer容灾体系包括以下几个方面(如图2-5所示)。

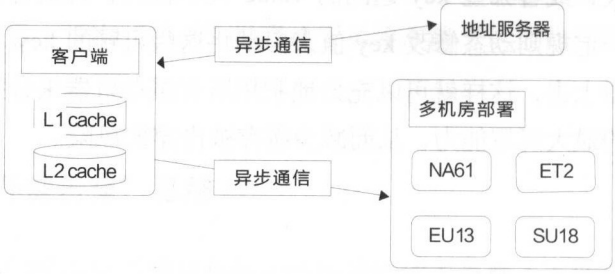


图 2-5 VIPServer 容灾体系图

（1）缓存机制

VIPServer 客户端有两级缓存机制保证在服务端不可用时，仍然不影响业务基本的服务。两种缓存机制分别是内存缓存（L1 cache）和磁盘缓存（L2 cache）。客户端在运行期间，从内存缓存中读取数据返回给应用；磁盘缓存用在客户端重启时，把之前内存缓存中的内容重新 load 到内存，形成内存缓存。

（2）异地容灾机制

为了应对单机房部署在机房断网时 VIPServer 服务端不可用的问题，VIPServer 采用多地多机房部署，保证一个或者多个机房出现故障时，不影响 VIPServer 的正常功能。

（3）异步更新机制

客户端和服务端的交互采用全异步的机制，确保不会因为服务端的问题阻塞业务应用。

（4）重试机制

客户端和服务端通信时，如果某个服务端出现异常，客户端自动重启其他机器，直到更新成功或者轮询完所有服务端。

然而，虽然 VIPServer 有着较为完善的容灾机制，但是还缺少另一个重要环节：限流保护。因为缺少保护，在异常流量很大时，VIPServer 整体还是出现了问题。

导火索

某日，VIPServer 上游的一个平台向它发送了一个修改配置的请求，但是配置修改之后，这个配置其实是不正确的，导致 VIPServer 客户端不能更新域名的 IP 列表。由于前面容灾体系中的重试机制，VIPServer 客户端在更新不到数据时会不断重试，直到轮巡完所有可用的 VIPServer 服务端机器，导致 QPS 暴涨，且正好这个域名的调用方很多，有几千个客户端调用，更加剧了 VIPServer QPS 暴涨，rt 被涨到 2s 以上，VIPServer 集群渐渐被拖垮。

连锁反应

阶段 1

VIPServer 开发在收到 rt 拉长的报警之后，开始执行 rt 变长预案：分批重启集群。但是这时候有一个问题，重启的一部分机器分布在张北和深圳机房，当时 VIPServer 的所有数据存放在中心 diamond，数据拉过来需要很长时间，在数据完全准备好之前，nginx 起来且 80 端口打开了！

阶段 2

此时，找不到服务出口的客户端请求，总算找到了一些可以服务的机器了。大量请求瞬间涌入这批刚刚启动服务的机器，服务端此时并没有客户端需要的数据，它会给客户端返回“dom not found”的信息。

阶段 3

一些老版本的 VIPServer C 客户端在收到返回 server 返回的“dom not found”的信息之后，会删除磁盘和内存的数据，而开发小成所支持的系统正好用的是这一版本的 C 客户端。由于 C 客户端已经没有域名的 IP 列表等信息，于是导致了应用调用方无法调服务的结果。整个过程如图 2-6 所示。

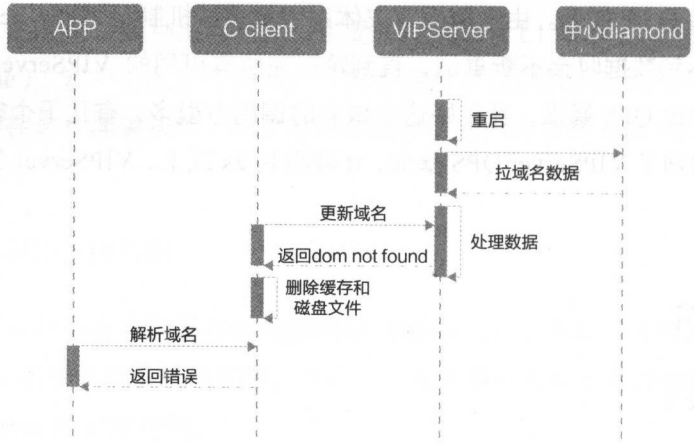


图 2-6 调用错误图示

细节分析

第一次重启时，rt 飙升原因还没有找到。但开发人员首要考虑的是如何快速消除影响，然后再来分析具体原因。

在当时的情景下，需要做的就是让开发小成的那部分应用快速拿到域名的 IP 列表。所以采取了另一种方案：人工重启服务，确认数据完整之后再打开 80 端口。按这种方案重启服务之后，客户端已经可以更新到 IP 列表，业务也开始慢慢恢复。

整个问题是由一个小小的配置错误引发的一系列的连锁反应导致的，暴露出的问题引起了相关方的反思。

- 预案缺乏演练，预案只有在真正发生问题时才会使用，平时基本不会演练。这就使得即使预案有问题，也发现不了。
- 考虑到是内部系统，当时 VIPServer 没有做好自身的限流保护，导致调用量暴涨时自己被压垮。
- 多个客户端间行为不一致，整个过程中，Java、DNS-F、Tengine 客户端依靠自身的容灾能力，仍然能够正常提供服务。但是某几个版本的 C 语言客户端因为业务方原因而牺牲了容灾能力。
- 系统太依赖自动化，缺乏出现问题时人工干预的方案。
- 应用平时应该经常梳理自己依赖的系统出问题时对自己的最坏影响是什么，如果不能接受这个结果，要考虑如何接入能把损失降到最低。
- 建立好上下游沟通方式，如把各个业务方的接口人组织到一起建立应急小组，发生故障时第一时间通知到各接口人，以最快的速度让大家知道哪个应用出了什么问题，降低大家排查问题的时间。

解决方案

针对上面的反思，考虑了以下解决方案。

- 跟故障演练平台（见 5.3 内容）合作，演练常见的故障，并根据演练情况制定故障预案。

- 服务端加上限流方案，针对域名粒度、IP 粒度和单机总 QPS 都做了限流。
- 梳理现有客户端的核心行为，核心行为保持一致。
- 客户端增加人工干预的方案，确保服务端全部有问题时可以人工介入快速消除影响。

小结

实现一个稳定可靠的分布式系统，需要我们考虑到方方面面，包括系统的容灾体系设计、部署架构、多种客户端的行为一致性、限流保护、紧急状态下人工干预方案、故障演练等。总结起来有以下几个原则。

（1）容灾体系保证当后端服务器挂了或者响应变慢时业务不受影响。

（2）部署架构上保证多地或者多机房情况下的高可用。

（3）客户端行为一致保证服务端某一变更，所有客户端的行为跟预期一致。

（4）限流保护保证流量异常或者应用大量扩容时，服务不被打垮。

（5）人工干预方案保证在服务端长时间无法恢复的情况下，人工纠正客户端的某些错误行为。

（6）故障演练验证上述方案的实际效果是否和预期一致，通过故障演练还可以不断发现新的风险点，验证并制定修复方案。

总之，系统的稳定性体现在一点一滴上，任何一种改动和方案都应该充分考虑和验证。

2.3　机房容灾，VIPServer 软负载流量调度实例

引子

VIPServer 是阿里巴巴内部广泛使用的软负载系统，主要用来解决部署在机房内或机房间的集群间的 http 流量负载调度。想了解 VIPServer 可以参考 CSDN 的文章《VIPServer：阿里智能地址映射及环境管理系统详解》。

本文的内容涉及机房接入层与对应的软负载产品，下面先介绍下相关的一些背景知识。

背景

硬负载与软负载

典型互联网系统架构中，机房的最前端，都会部署一套负载均衡设施，用来分发入口 http 请求到后端提供实际服务的应用集群，海量用户场景下我们希望这套负载均衡与后端集群机器都可以各自方便地横向扩展。

如图 2-7 所示，硬负载以 F5、Array 为代表，其设计兼顾高可用一般会做硬件冗余，负载分一主一备，备机平时没有流量，这种专有硬件成本高且会被厂商技术绑定，横向扩展能力一般有限。

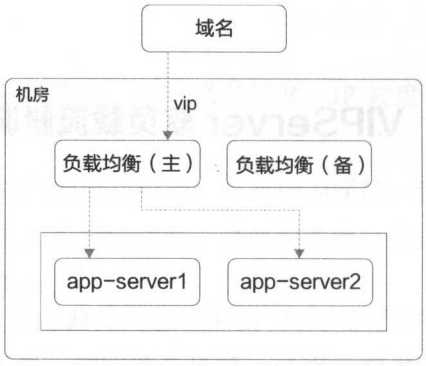


图 2-7 机房硬件负载均衡部署架构

基于 Linux 体系的软负载产品只需要普通服务器，对于互联网技术体系来说更适合。常见可用于软负载的产品有 Nginx(HTTP)、HAProxy（TCP/HTTP）、LVS(TCP)。

如图 2-8 所示，软负载产品有个特点：都是中心化的，流量都要过一次软负载后进行转发。

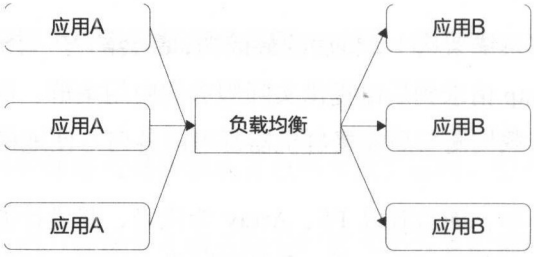


图 2-8 中心化负载均衡

中心化负载均衡在大流量及并发场景下有成为性能隐患和故障单点的风险，试想以阿里巴巴体量下的用户访问流量，在机房入口进来，走中心化负载均衡后再重新流入到各个 web 集群，这样的路径是否合理？

再打一个通俗的比喻：一个大坝泄洪，出来的洪水如果先进了一个小

水库，再重新从这个小水库流到河的各个支流里，先不考虑这个小水库会不会被冲垮（风险很高），一个正常的思路是不是应该去掉这个水库直接泄洪到下游支流上更合理？

而 VIPServer 希望解决的就是此种问题，其设计上采用的是典型分布式服务注册与发现的思路，客户端在获取了 http 服务的地址后，直接发起调用，越过了中心负载。

如图 2-9 所示，去网关化的软负载形式，往往会将负载均衡策略下沉到客户端（中心已经没有了，无法在中心进行统一的负载均衡策略设置），业内同类产品 Netflix 的 Eureka 也采用了同样的思路。

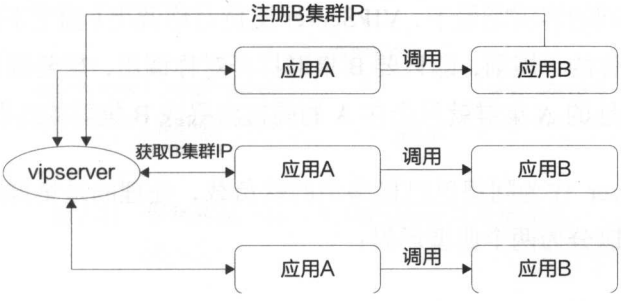


图 2-9　去网关化的负载均衡

接入层与软负载

大型互联网架构中，会以微服务的形式拆分应用，此外仍会有横向的关注点，比如：https 证书卸载、网站防爬防攻击等安全策略、统一的负载均衡策略、收敛公网 IP 入口、平滑扩容。

前三个需求没有必要在下层的应用层每个都各自实现一次，统一接入层就是为了解决这个问题，接入层通常会选用反向代理实现。

如图 2-10 所示，是一个简单的单机房部署结构示例，一般稳定性要求

较高的业务会采用同城双机房部署的方式保障可用性。

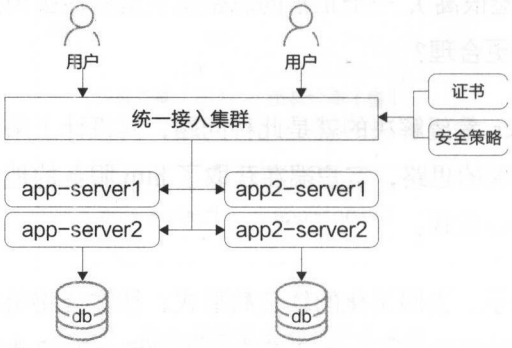


图 2-10 单机房部署结构示例

在机房间的容灾场景下，VIPServer 支持对称调用（指上下游调用在机房层面是对称的）规则，即只要 B 集群打开对称调用，想要调用 B 集群提供的 http 地址的 A 集群就只会在 A 的同机房寻找 B 集群的机器。

VIPServer 作为阿里巴巴机房用的软负载，处理的是集群间内网 http 的调用。可以分为两个典型场景：

第一个场景是接入层发现下层 web 服务时的负载均衡。

第二个场景是下层 web 服务集群间的负载均衡，web 应用集群间在机房内网调用流量（域名方式）不用去机房外公网走一圈再回到内网的另一个集群，减少性能损耗。

统一接入作为机房流量的入口，其集群是 VIPServer 最大的客户端，所有接入统一接入的 http web 应用，都需要去 VIPServer 控制台注册一个 key（通常用域名标识加后缀，好辨识）后接入。

如图 2-11 所示，统一接入的 Tengine 上配的模块会从 VIPServer 服务端取对应的 web 应用 server 地址，统一接入的 VIPServer client 模块与通用

的 VIPServer client 不同，有自己的定制策略，即接入了统一接入的上游 web 应用集群，默认都会开启对称调用，会优先在同一机房寻找 upstream web 应用。

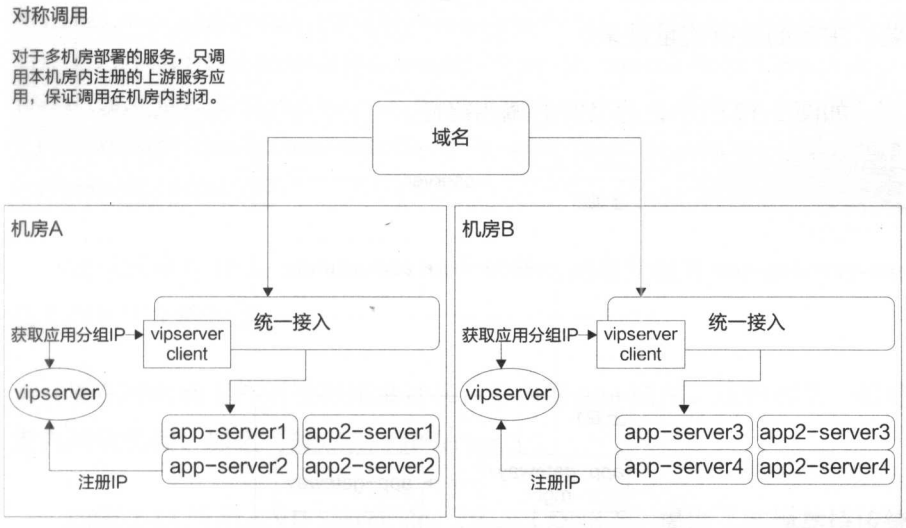


图 2-11　多机房统一接入与 VIPServer 及对称调用

实战案例

了解了软硬负载的背景知识后，本文将从使用方角度结合实际案例介绍 VIPServer 工作的行为。

业务有两个应用，应用间的调用关系是这样的：

app-gateway-mgt（部署机房：上海 1）调用 app-gateway（部署在两地三机房：上海 1/上海 2 / 深圳 1），app-gateway-mgt 是管理后台，app-gateway 是被前者管理的 API 网关。管理后台会对网关发起请求。

问题现象

app-gateway 的深圳机房刚部署，不希望有流量，在应用启动后，发现深圳机房有日志输出并报 FAIL_SYS_API_NOT_FOUNDED，报什么不重要，主要的是有流量进来。

如图 2-12 所示，为当时的调用路径。

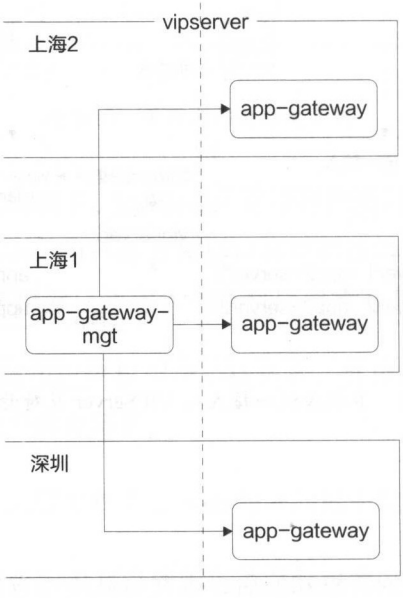


图 2-12 应用跨机房调用

为了不影响线上，先把深圳机房的应用杀掉再查原因。

疑问：流量从哪里来的？

先看域名指向，dig 一下 app-gateway 的域名，可以看到域名 CNAME 到上海统一接入，没有深圳统一接入。那么外网的流量是不会从深圳统一接入层流入深圳的 app-gateway 机器的。

查看 app-gateway 深圳机器的请求日志 mtop-open-monitor.log 内容，如下。

```
20xx-xx-xx 23:47:22,^app
-gateway.unsz.xxx@10.15*.18*.17*,^mopen.xxx.get,^1.0,^10.*.*.168,^1492703
239,^21272243,^122c59fbb46049ae9b4add36773e013f,^Apache-HttpClient/4.5
(Java/1.8.0_66),^,^11320,^taobao,^android,^5.0.0,^1,^132,^FAIL_SYS_API_NO
T_FOUNDED::<C7><EB><C7><F3>API<B2><BB><B4><E6><D4><DA>,^GW,^sys_fail,^0,^
,^,^Degarde,^ic1000002,^
```

发现调用方 IP 是 10.*.*.168，这个机器是调用发起方 app-gateway-mgt 在上海机房 1 的机器。

现在可以确定外网没有流量过来，是内网 http 调用，这种调用一般是走内网软负载，那么可疑点在 VIPServer 上。

如图 2-13 所示，VIPServer dns client（DNS-F）提供了本地劫持传统域名的功能，调用方机器在发起 http 请求时会先解析本地的 DNS，这时如果安装了 dns VIPServer DNS Client，会优先走 VIPServer 的 DNS 缓存。

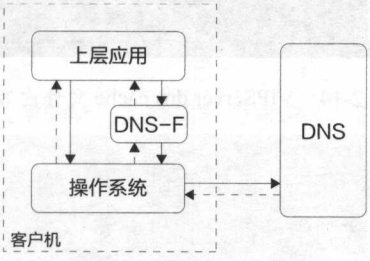


图 2-13　DNS-F 工作示意图

VIPServer DNS Client 的 RPM 包名是 vipserver-dnsclient。

排查调用发起方 app-gateway-mgt

既然怀疑是 VIPServer 把流量分过来的，那么先看下这个应用有没有装 vipserver-dnsclient。在应用基线上没有发现该安装包，再根据 VIPServer 文档给的安装路径到线上问题机器查看，发现了安装目录是存在的。

VIPServer Client 的域名缓存目录在 /xxx/vipsrv-dns/vipsrv-cache,缓存域名的文件为 xxx.taobao.com.cache。

如图 2-14 所示，看看 xxx.taobao.com.cache 中保存了什么？

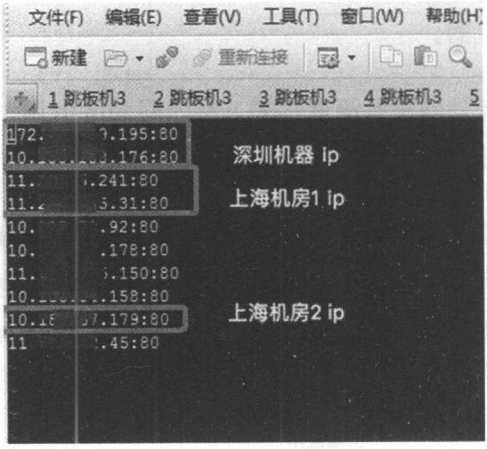


图 2-14 VIPServer dns cache 文件内容

果然缓存了下游应用三个机房的机器！为什么有三个机房呢？因为 xxx.taobao.com 的 vip 域名申请时没有打开对称调用，VIPServer 服务端会返回给 client 端所有机房的机器 IP。

如图 2-15 所示，此时 VIPServer 配置情况是这样的。

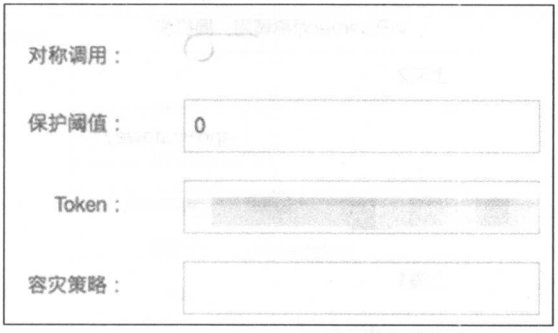


图 2-15　VIPServer 策略

开启对称调用

技术人员需要上海机房 1 的上游机器只调用上海机房的机器，那么需要打开下游 VIPServer 端的对称调用规则。

如图 2-16 所示，开启了对称调用同机房策略后，上海机房 1 的 VIPServer Client（也就是 app-gatewaymgt）应该只能默认拿到同机房上海机房 1 的下游应用（app-gateway）IP，再看下 VIPServer 本地 DNS 缓存内容，只剩下上海机房 1 的机器了。

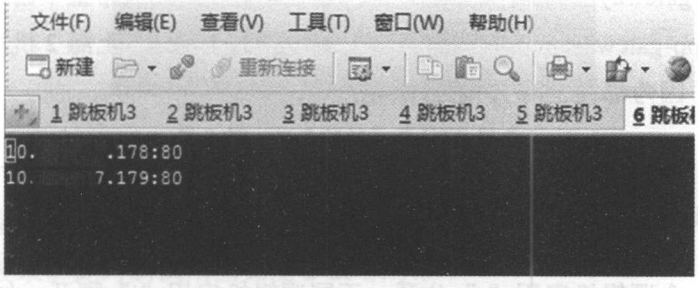


图 2-16　开启对称调用后 VIPServer dns 内容

此时的调用图如图 2-17 所示。

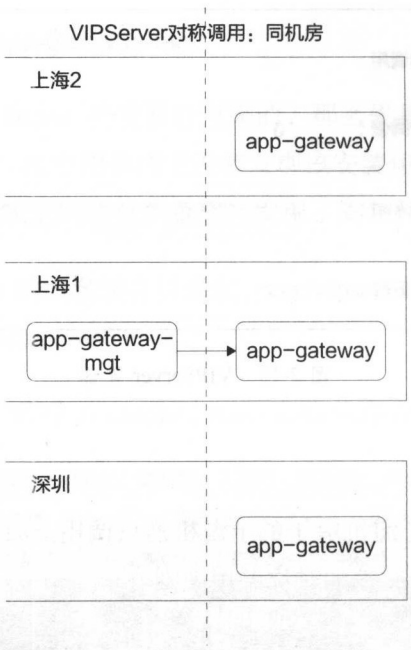


图 2-17 开启对称调用（机房未归组）

设置机房归组

app-gateway 上海机房 2 还有好几台机器，技术人员希望下游上海机房 1 里的 app-gateway-mgt 在调用时将上游应用的上海 1、上海 2 机房视为一个机房（因为都在上海），怎么做呢？

VIPServer 还提供机房归组功能，强行将我们需要的机房合成一个逻辑分组，我们要的就是上海两个机房变为一个逻辑分组：

同一个逻辑机房用“,”分开，不同逻辑机房用“|”隔开，如图 2-18 所示的例子是指上海 1，上海 2 为一组，深圳单独一组。

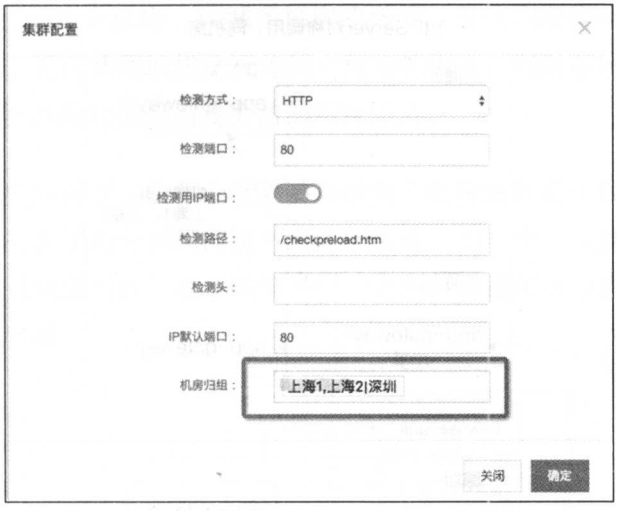


图 2-18　设置对称调用并同城机房归组

设置好后等待 VIPServer 推送，如图 2-19 所示，看下本地 VIPServer DNS 缓存。

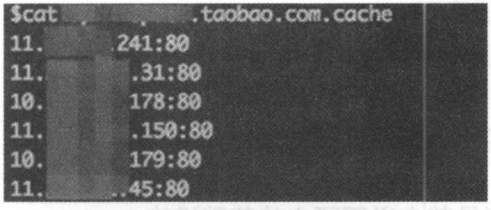


图 2-19　机房归组后的 DNS 缓存

在运维平台上通过机器 IP 查询，都是上海 1、上海 2 机房的机器，完全符合要求。此时的调用图如图 2-20 所示。

至此，可以看到 VIPServer 软负载可以根据情况灵活调度同城双机房及异地机房的 http 流量，解决业务上的容灾及性能需求。

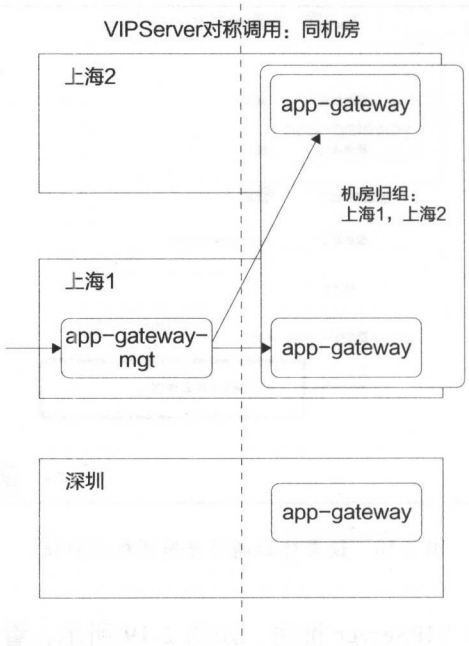


图 2-20 VIPServer 对称调用并机房归组示意图

小结

Q：互联网系统如何做到高可用？

A：负载均衡是其中必不可少的基础设施。

Q：那么负载均衡如何帮助上游应用高可用？

A：系统容量是否已经达到了可处理的警戒线，每天线上运维都需要关注。负载均衡既要考虑自身支撑的外部流量是否已经到达阈值，也要考虑部署在上游的应用承载能力以及健康状况。

Q：负载均衡设计上都要考虑什么？

A：基于异地多机房、同城多机房的场景，负载均衡需要考虑同城双机房延时低，可以将同城机房视为同一机房的需求；同时还能将异地机房作为本地发生故障时流量自动分流的可选方案。

同城双机房场景，负载均衡需要考虑两个机房的容量承载。如图 2-21 所示，如果打开了对称调用同机房优先的规则，当一个机房的若干台机器故障已经超过阈值可能无法承载服务时，要能将流量部分分流到另一个机房帮助承担负载。

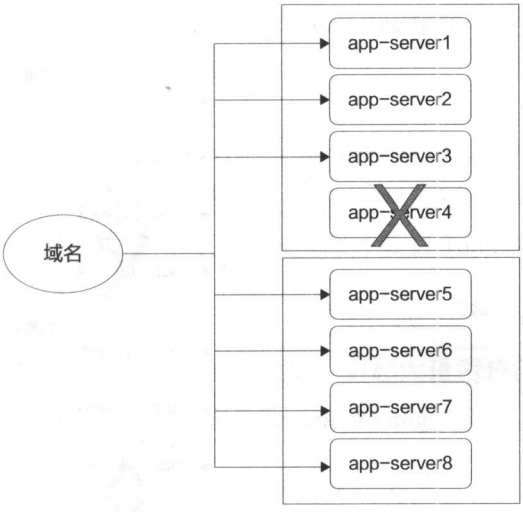


图 2-21　对称调用超过阈值则转发其他机房

在以上机房重新负载的情况下，还要考虑一种极端情况，这种策略是全局的。当故障机出现，流量已经根据机房维度进行了重新负载，但情况恶化，故障机器还在增加，剩余的机器已经无法负担整体的流量时，分布式系统就进入了跛脚鸭模式，剩余的机器负载过高，rt 越来越长，一台接一台。此时系统必须用有损的方式进行降级了，需要舍弃一部分流量，让剩下的机器刚好能处理当前负载的一部分。那么负载均衡应支持一种全局阈值，当总机器低于一定数量时，重新将流量分流到那些已经故障的机器

上，以损失掉这部分流量为代价，让剩余的机器能处理其能处理的请求。如图 2-22 所示。

运维友好设计上，负载均衡需要实时感知后端机器的上下线情况，并将下线机器从服务列表中摘除。这种情况与对后端健康监测的摘除不同，下线机器的实时信息不是从负载均衡自身的健康监测得到的，而是从其他运维系统的 CMDB 信息中得到的。同时有简洁的控制台能让开发迅速了解当前集群的健康状况以及服务节点的更新情况，能方便自助修改集群的容灾策略。

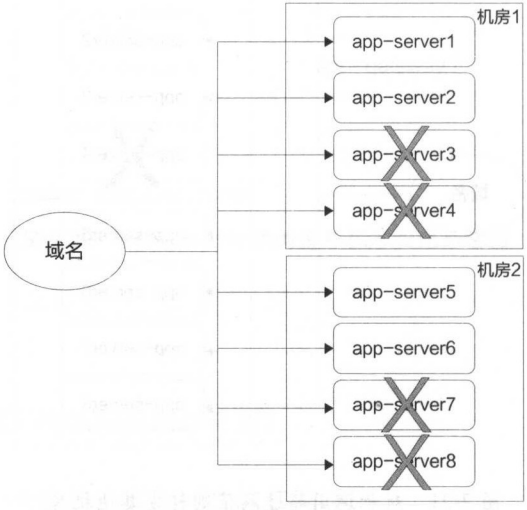


图 2-22 全局保护——有损降级

以上的设计考量可以作为抛砖引玉，读者还可以想想是否还有其他约束，以及软负载产品自身作为基础设施该如何保证高可用。

2.4　山洪暴发，高流量触发 Tomcat bug 引起集群崩溃

背景

中间件 M 应用在 2013 年开始采用 Async Servlet+HSF Callback 调用模型，以提高其性能，从而获得较好的效果。当时 Tomcat 的连接协议，采用 APR 模式，但 2013 年双 11 前，发现 APR 的 Tcnative 存在一个导致 JVM Crash 的 bug，提交了 bug 给官方。

在 2014 年阿里巴巴集团容器团队升级 Tomcat 7.0.26 到 7.0.54，同时经过压测比对和对 Tcnative 的维护成本等考虑，决定替换 APR，采用 NIO 模式，2015 年第一季度，M 应用线上机器全部升级到 Tomcat7.0.54，同时采用 NIO。

2015 年×月×日，网络上做了个变更，引起大量用户请求重试，形成了几倍的流量高峰，高流量下触发了 Tomcat 的 bug，导致了整个应用集群崩溃。

NIO 模式背景介绍

Tomcat 共有三种连接器模式，BIO/NIO/APR/NIO2，其中 NIO 是异步 IO 模式的简称。在默认的配置下，NIO 的模式的实现模型如下。

- Acceptor 线程：全局唯一，负责接受请求，并将请求放入 Poller 线程的事件队列。Accetpr 线程在分发事件的时候，采用 Round Robin 的方式来分发。
- Poller 线程：官方的建议是每个处理器配一个，但不要超过两个，

由于现在几乎都是多核处理器，所以一般来说都是两个。每个 Poller 线程各自维护一个事件队列（无上限），它的职责是从事件队列里面拿出 socket，往自己的 selector 上注册，然后等待 selector 选择读写事件，并交给 SocketProcessor 线程去实际处理请求。

- SocketProcessor 线程：Ali-tomcat 的默认配置是 250（参见 server.xml 里面的 maxThreads），它是实际的工作线程，用于处理请求。

一个典型的请求处理过程

如图 2-23 所示，是一个典型的请求处理过程。其中深色框代表线程，无填充色框代表数据。

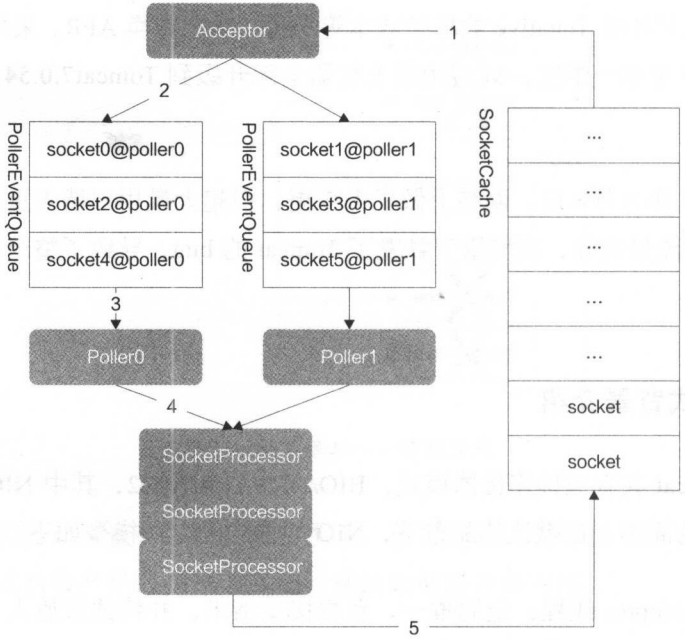


图 2-23 一个典型的请求处理过程

Acceptor 线程接受请求，从 socketCache 里面拿出 socket 对象（没有

的话会创建，缓存的目的是避免对象创建的开销），Acceptor 线程标记好 Poller 对象，组装成 PollerEvent，放入该 Poller 对象的 PollerEvent 队列，Poller 线程从事件队列里面拿出 PollerEvent，将其中的 socket 注册到自身的 selector 上，Poller 线程等到有读写事件发生时，分发给 SocketProcessor 线程去实际处理请求，SocketProcessor 线程处理完请求，socket 对象被回收，放入 socketCache。

原因定位

本次问题的触发，是瞬间大流量上来，Tomcat 直接处于假死状态，不再处理请求，请求失败引起了大量的用户重试（这也是常规表现），问题表现是高 QPS 流量压力下，M 应用机器无法正常服务。

当时，开发人员首先想到在茫茫日志当中，grep 一个 ConcurrentModificationException 的异常，果然发现这个异常信息：

```
   Exception in thread "http-nio-7001-ClientPoller-1"
java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(HashMap. java:793)
        at java.util.HashMap$KeyIterator.next(HashMap.java: 828)
        at java.util.Collections$UnmodifiableCollection$1.
next(Collections.java:1010)
        at org.apache.tomcat.util.net.NioEndpoint$Poller.
timeout(NioEndpoint.java:1421)
        at org.apache.tomcat.util.net.NioEndpoint$Poller.
run(NioEndpoint.java:1215)
        at java.lang.Thread.run(Thread.java:662)
```

技术人员都很熟悉这个异常，并发操作 HashMap 的错误，因为之前已经发现过这个问题，当时也向 Tomcat 提交了 bug，只是排查了很久没有定位到原因，连 Tomcat 的大神 Mark Thomas 也仔细分析过，当时准备的解

决方案是先捕获这个异常（参考资料：https://bz.apache.org/bugzilla/show_bug.cgi?id=57943），报了这个异常后的表现还是非常清晰的。

这个异常没有被捕获，Tomcat NIO 的 Poller 进程会退出，熟悉 Tomcat 的人都知道（如上介绍 Tomcat NIO 的原理），Tomcat 有 2 个 Poller 线程，请求被 Accept 线程接受后，会被分配到这两个 Poller 线程进行 NIO 的监听，一个线程退出了，就形成了个黑洞，那个存活的 Poller 线程可以正常处理业务，而分配到挂掉 Poller 线程的连接请求都无法处理。

另外 Tomcat 有个连接计数器，maxConnections，NIO 下默认是 10000，因为挂掉的 Poller 线程上的连接请求一直不处理，所以随着时间累计，分配到这个线程上的连接越多（正常的 Poller 处理完连接后，每次都有 50% 连接被分配到挂掉的 Poller 线程，挂掉的 Poller 线程成为黑洞，慢慢吸收连接过去），直到挂掉的 Poller 线程占满 10000 个连接，Accept 线程发现连接已满，拒绝接受新的请求，Accept 线程进入 locksupport.park 状态，因 Accept 线程停止工作，TCP 全连接队列满，再引起半连接队列满，最后 TCP 不再响应 TCP 连接。开发人员看到的就是 TCP 握手发数据包无响应，M 应用彻底停止服务。

细节分析

现在的问题很清晰，为什么会有并发操作 HashMap 的问题，而且这是已知的问题，只是一直没有进展，是否捕获这个异常后就全部正常呢？开发人员与 Tomcat 团队一起进行了深入的分析。

根据异常堆栈，可以知道，Poller1 线程在处理超时的时候，会遍历 selector 上的 keys，然后在这个遍历的过程中，另外一个线程修改了 keys 的数据，导致抛出异常。大致代码如下：

```
protected void timeout(int keyCount, boolean hasEvents) {
    ...
    // 超时
    Set<SelectionKey> keys = selector.keys();
    ...
    for (Iterator<SelectionKey> iter = keys.iterator(); iter.hasNext();) {
      SelectionKey key = iter.next();
      ...
    }
    ...
```

代码分析看不出端倪，所有对 Selector 的操作都是在 ClientPoller 的线程上做的修改，而每个 ClientPoller 在初始化的过程中各自持有自己的 Selector，设计上决定了不应该有多线程的问题。那么到底是谁在修改底层的 keys 呢？

由于在 Tomcat 源码中看不出问题所在，开发人员决定在 JDK 层面打 patch，在 JDK 层面，所有可能修改 keys 的地方只有两处，sun.nio.ch.EpollSelectorImpl 的 implRegister 方法以及 implDereg 方法（注意，不同操作系统下的实现类可能不一样，在 linux 操作系统下是 sun.nio.ch.EpollSelectorImpl）。

开发人员下载了 openjdk 源码，在 sun.nio.ch.EpollSelectorImpl 的 implRegister 方法以及 implDereg 方法中打印相关信息，并通过以下的方式覆盖 JDK 的原有实现：

```
JAVA_OPTS="$JAVA_OPTS -Xbootclasspath/p:/var/tmp/ selector-patch.jar"
```

不幸的是，加上 patch 之后问题无法重现，看来输出 debug 信息本身拖慢了线程的执行，导致本来就很小的时间窗口稍纵即逝。不断缩减输出内容也没有任何重现的迹象，怀疑是输出日志的开销影响了问题的复现，

当时打的日志比较多，每个日志比较详细导致信息输出量较大，因此不断重试，不断缩减日志，这是磨练耐心的过程，在不断重试过程当中，终于发现了一个信号：

```
http-nio-7001-ClientPoller-1> sun.nio.ch.EPollSelectorImpl@cc98f2c
http-nio-7001-ClientPoller-1> sun.nio.ch.EPollSelectorImpl@5787ae61
http-nio-7001-ClientPoller-0> sun.nio.ch.EPollSelectorImpl@cc98f2c
```

看这输出的日志信息，同一个 Selector 实例被两个 ClientPoller 线程上引用，这就很可能是并发的问题。

```
socket.getIOChannel().register(socket.getPoller().getSelector(),
SelectionKey.OP_READ, key);
```

对于这个现象唯一可能的解释是，当 ClientPoller 调用这行代码的时候，只有当 socket.getPoller().getSelector()获得的 Selector 实例和当前 ClientPoller 所持有的实例不一致，才有可能造成前面看到的 ClientPoller 会操作不同 Selector 实例的现象。

也就是说，socket.getPoller()拿到的 Poller 的实例，不是自己。

这怎么可能！因为 socket 所代表的数据结构 PollerEvent 是放在 ClientPoller 自身队列里，并且放进来的时候 PollerEvent 上的 Poller 实例会被置成当前的 ClientPoller。

```
public void register(final NioChannel socket) {
    socket.setPoller(this);
    ...
    addEvent(r);
}
```

那为什么当 ClientPoller 线程再从 socket.getPoller()得到的 Poller 实例

不是自己？为了验证这一个看似不可能的猜想，加入了以下的调试信息来观察当前线程 ClientPoller 和 socket 所持有的 Poller 的关系。

```
    if (socket.getPoller() != this) {
        System.out.println(">>> t: " + Thread.currentThread().getName() + ",
s: " + socket.getPoller().getName() + ", " + socket + ", ts: " +
System.currentTimeMillis());
    }
    socket.getIOChannel().register(socket.getPoller().getSelector(),
SelectionKey.OP_READ, key);
```

果然有收获。这说明了 socket 所持有的 Poller 在某些条件下不一定和当前的 ClientPoller 线程一致。

```
t: http-nio-7001-ClientPoller-1, s: http-nio-7001-ClientPoller-0
t: http-nio-7001-ClientPoller-0, s: http-nio-7001-ClientPoller-1
t: http-nio-7001-ClientPoller-1, s: http-nio-7001-ClientPoller-0
```

进一步的调试发现，当 Poller 线程从自身的事件队列中拿出 socket 对象的时候，socket.getPoller 拿到的实例就已经错乱了，但是 Acceptor 线程放入事件队列之前，socket.getPoller 拿到的实例是没有错乱的。那么可以肯定的是，socket.getPoller 拿到的对象在进入事件队列之后，被修改了。

为什么会被修改呢？出现这个情况只有两种可能：

- socket 对象本身被修改了。
- socket 里面的 Poller 对象被修改了。

socket 对象封装在 PollerEvent 对象中，如果是 socket 对象本身被修改，那么只有一个入口，就是 PollerEvent 的 reset 方法，在 reset 方法里面加入调试信息，却没有任何发现。这个方法在 socket 对象进入队列后根本没有调用过。

如果 socket 对象没被改变，那唯一的可能性就是 socket.getPoller 的对象会发生变化。也就是说**同一个 socket 有可能会被修改两次，而且还是来自两个不同的 Poller 线程**。

通过对 ClientPoller 上游的分析，发现 Tomcat 是这么工作的，当请求进来的时候，有一个 Acceptor 线程负责建立连接，然后递交请求给后面的两个 ClientPoller 线程来处理。

```
protected boolean setSocketOptions(SocketChannel socket) {
        // 对连接进行处理
        try {
            // 设置非阻塞，同 APR 处理模式，采用 poll 模式
            socket.configureBlocking(false);
            Socket sock = socket.socket();
            socketProperties.setProperties(sock);

            NioChannel channel = nioChannels.poll();
            if ( channel == null ) {
                // 加密 SSL 连接 channel 构造
                if (sslContext != null) {
                    ...
                    channel = new SecureNioChannel(socket, engine, bufhandler,
selectorPool);
                } else {
                    // 普通连接 channel 构造
                    ...
                    channel = new NioChannel(socket, bufhandler,
Thread.currentThread());
                }
            } else {
                channel.setIOChannel(socket);
                if ( channel instanceof SecureNioChannel ) {
```

```
                SSLEngine engine = createSSLEngine();
                ((SecureNioChannel)channel).reset(engine);
            } else {
                channel.reset();
            }
        }

        getPoller0().register(channel);
    } catch (Throwable t) {
        ...
        // 通知上层关闭 socket
        return false;
    }
    return true;
}
```

为了避免垃圾回收的压力，channel 并不是每次都创建，而会从一个 nioChannels 的缓存队列中获取，并在使用完毕后放回该缓存队列。在排除掉 nioChannels 的数据结构(ConcurrentLinkedQueue)本身的并发问题之后，唯一的可能性就指向了：在某种情况下这个缓存队列中，同一个 channel 对象在队列中存在多份。

按照这个思路，在 nioChannels.offer 以及 nioChannels.poll 之前判断队列中是否已经存在要还回的 channel 实例，或者有要还回的 channel 实例，如果有，就打印出以下信息。输出的结果证明了猜想是正确的，这是一个重大的进展。

```
   >>> poll but still found in queue,
org.apache.tomcat.util.net.NioChannel@28baf3bf:java.nio.channels.SocketCh
annel[closed], ts: 1435720813750
   >>> offer but already found in queue,
org.apache.tomcat.util.net.NioChannel@28baf3bf:java.nio.channels.SocketCh
```

```
annel[closed], ts: 1435720812885
```

但是这挡不住技术人员刨根问底的好奇心，我们还想知道，socket 对象到底在什么地方被重复放了两次，以及有哪些线程会通过 nioChannels.offer 还回 socket 对象。由于 nioChannel.offer 是发生在 SocketProcessor 的 run 方法里，并且 run 方法的总入口是：

```
    public boolean processSocket(NioChannel socket, SocketStatus status,
boolean dispatch, int where) {
            ...
            SocketProcessor sc = processorCache.poll();
            if ( sc == null ) sc = new SocketProcessor(socket,status, where);
            else sc.reset(socket,status, where);
            if ( dispatch && getExecutor()!=null )
getExecutor().execute(sc);
            else sc.run();
            ...
        }
```

通过改造代码打标(where)并输出当前执行线程，最终得到以下的日志：

```
    thread: Thread[http-nio-7001-ClientPoller-1,5,main], key: , ka: null,
where: 6
    thread: Thread[http-nio-7001-ClientPoller-1,5,main], key:
sun.nio.ch.SelectionKeyImpl@7da286bb, ka: null, where: 1
    thread: Thread[HSF-CallBack-11-thread-2,10,main], key:
sun.nio.ch.SelectionKeyImpl@7963cb1e, ka: null, where: 10
```

其中标记 1 是 Poller#processKey，标记 2 是 Poller#timeout，标记 3 是 Http11NioProcessor#actionInternal（注意，这里的线程是 HSF-CallBack 线程，不是 http-nio-7001-ClientPoller 线程）。其中唯一有可能并发的地方就是 HSF-Callback 线程与 http-nio-7001-ClientPoller 线程在交还 nioChannel

的时候并发，并导致错误。通过阅读代码，发现 HSF-CallBack 线程这个时候在做 Async Complete 的操作。

解决方案

在 Tomcat 7.0.58 版本以后，官方针对一个类似的对象池污染的问题（Bug57340）做过一个修复。大致的逻辑是通过原子操作来保证多个线程并发的时候最终一个 socket 对象被 offer 进入缓存队列。事例代码如下：

```
public void run() {
    SelectionKey key = socket.getIOChannel().keyFor(
                socket.getPoller().getSelector());
    ...
    doRun(key, ka);
}

private void doRun(SelectionKey key, KeyAttachment ka) {
    int handshake = -1;

    if (key != null) {
        handshake = ...
    }

    if (handshake == 0) {
        // 官方针对bug57340的修复在这里
    } else if (handshake == -1 ) {
        if (key != null) {
            socket.getPoller().cancelledKey(key, SocketStatus.DISCONNECT,
false);
        }
        nioChannels.offer(socket);
        socket = null;
```

```
            if ( ka!=null ) keyCache.offer(ka);
            ka = null;
        }
    }
```

虽然说升级到 7.0.59 后并没有出问题，但是分析发现，在 else if (handshake == -1)分支中，也有可能出现重复放入的情况：

```
>>> 2. thread: Thread[http-nio-7001-ClientPoller-0,5,main], key: , ka:
null, where: 1
```

出现这个情况也是可以理解的。handshake 默认值是-1，其中的一种可能性就是入参的 key 为空，很显然，代码没有考虑到处理这种情况。当 key 为空的时候，从 run 方法的第一行代码可以知道，这个时候 socket 已经被其他线程处理过了。

也就是说，官方在 7.0.58 上提供的 Fix 并不能完整的修复这个问题。

为了避免多线程不往同一个队列里重复放同一个对象，有以下几个方案：

（1）在队列放入队列之前检查是否有重复，如果有重复，就不放入，优点是实现简单明了，缺点是必须要加锁，这会极大降低并发度。

（2）在多线程放入队列之前，保证只有一处地方放进去了。好处是，缓存队列还是无锁设计，不会降低并发度。缺点是开发难度比较高，必须要保证所有可能涉及的地方都要修改。

（3）放弃缓存，不再重用对象。这样能根本解决这个问题，但是缺点也是显而易见的，会带来极大的 GC 开销。

官方在 7.0.58 上的做法，实际上是方法二，但是正是因为采用这种方

法，导致考虑并不周全，造成遗漏。

对于遗漏的地方，团队内部采取的办法是方法二和方法三中间的折中方案，即在 key 不等于 null 的时候，再放回缓存队列，对于返回值为 null 的情况下，不再放入缓存，而是直接丢弃。这是一种在重用和垃圾回收开销之间的权衡方案。

然而到这一步真的就完了吗？并没有！

代码里面不仅仅有 nioChannels 这个缓存队列，还有以下其他几个缓存队列：

- keyCahce
- processorCache
- eventCache

那么这几个队列里面有没有可能同样有对象重复放置的问题呢？

答案是有的。keyCache 就有这个可能。原因很简单，它的放回逻辑跟 nioChannels 完全一致。所以 keyCache 的放回逻辑也应该控制起来，避免重复 offer。

最终的修复方案如下：

```
                    }
                }
            } else if (handshake == -1 ) {
+               // 如果 key 为 null，则直接丢弃 socket，否则我们需要在很多地方加锁控制保障线程安全
+               // 同时还需要在 if 的条件当中有插入缓存队列逻辑，重用和性能间权衡吧
                if (key != null) {
```

```
-                    socket.getPoller().cancelledKey(key,
SocketStatus.DISCONNECT, false);
+                    if (socket.getPoller().cancelledKey(key,
SocketStatus.DISCONNECT, false) != null) {
+                        nioChannels.offer(socket);
+                        if (ka != null) keyCache.offer(ka);
+                    }
                 }
-                nioChannels.offer(socket);
                 socket = null;
-                if ( ka!=null ) keyCache.offer(ka);
                 ka = null;
             } else {
                 ka.getPoller().add(socket, handshake);
             }
```

经过验证，问题已经不再发生，开发人员逐步发布上线，同时也向 Tomcat 提交反馈 patch，很快官方也修复了这问题。

小结

（1）依据墨菲定律，凡是可能出错的事有很大几率会出错，本文案例表现为在 Tomcat 切换 NIO 后一个月已经发现个别机器发生错误，而且开发人员也进行了排查，也有临时解决方案在灰度当中。但是一次网络变更后大范围地触发了这个问题，所以任何的细小问题都不可以忽视。

（2）既然使用 Tomcat，那就要对其实现原理有比较清晰的了解，出现问题不能排除任何可能，包括 Tomcat、JVM、Linux 等处理请求的完整链路，从网络、硬件、内核、软件整体去排查分析。

（3）大集群带来了很多优势，但随着流量的上涨，也形成了一个巨大

的单点，后续应用的去中心化方案实施落地，也是针对这个点展开。

（4）高并发的问题的复杂性导致排查难度很大，但没有捷径可走，更需要耐心、细心。本文一些方法可供参考。

第3章 数据库常见问题

阿里巴巴集团数据库的演变经历了从 MySQL 到 Oracle 又回到 MySQL。再次回到 MySQL 时，架构已经从原来的单一的集中式数据库变为分布式数据库。

数据库架构的演变，给数据库运维和开发带来了巨大的挑战，也催生了数据库运维和开发自动化产品。此外，在数据库运维产品采集的性能数据和全量 SQL 数据的基础上，数据库产品团队结合阿里 DBA 多年积累的性能诊断经验开发了一个数据库智能诊断引擎。而数据库异地多活能力建设的过程，又催生了数据传输产品，提升异地数据同步性能和提高异地多活建站效率。

阿里的业务、数据库自动化运维产品、自助开发产品以及数据传输产品，还有其他数据库下游产品等都对 MySQL 数据库提出非常高的要求，如提升连接池的性能、单行并发更新能力、降低一些场景下的锁冲突概率、将 memcached 引擎引入到 MySQL 内核中等。阿里的 MySQL 内核团队实

现了一个又一个独特的功能，解决了业务一个又一个痛点，同时也在全球范围内提升了 MySQL 使用的水平。比如说 2016 年，AliSQL 的单行并发更新的 TPS 能到 9.4 万，这个能力就是阿里双 11 大促的核心竞争力之一。

所有这些巨大的进步都不是一蹴而就的，产品都经过了多次迭代更新，业务在使用过程中也多次试错，付出了一些代价。本章就是从开发角度选取了几篇典型的案例分析供读者借鉴。

3.1 性能杀手，SQL 执行计划

背景

某日上午，小集购买 a 产品失败，页面弹出“系统异常，请稍后重试”的报错，便联系了技术团队的开发小成。

“小成，我刚才尝试买 a 产品一直显示系统异常，是不是有什么问题呢？”开发小成接到电话后，迅速着手排查，同时也收到了系统监控平台的告警。

小成立刻登录系统应用服务器，发现交易耗时在 8s 左右。经过一系列紧张的故障排查和恢复工作，虽然过程艰难，但最后还是成功地恢复了业务。

解决过程

【DAY1】

“交易的耗时为什么会这么久？是不是内存资源不足引起的？”

小成立即查看了应用服务所在的公有云服务器的资源状态，发现服务器的 VM 系统大量 OOM 报错，即使尝试重启 Tomcat，也是失败。

“是不是有什么业务层的程序大量消耗资源导致？”

带着这个问题，小成继续一步步排查。半小时左右，小成发现合作机构的前置 T 应用上有大量的 Close_wait 连接，应用提示 open too many files。大量 Java 的应用报错 Java.io.IO.Exception: Broken Pipe.

为了尽快恢复业务，小成尝试重启了 Tomcat 应用，但错误依旧存在，尝试修改 open files 和 max user processes 参数，未见明显效果。小成又再试着在弹性服务器上分别 Telnet 本地端口和对方端口甚至重启了服务器。这些都无济于事，错误依然存在。

发现这些常用恢复方式不起作用，小成调整思路，看业务平台是不是有问题。果不其然发现应用平台上持续大量业务报错，报错信息集中在产品中心的捞取数据同步的定时任务。

“难道是这个定时任务的数据量增大导致的？”

小成确认了当日的定时同步数据的任务业务量级，发现每分钟 1 万左右，平均每秒（QPS）300~350 之间（与日常调用量持平）。

“既然定时任务的量级未变，为什么会大量报错？”

业务恢复争分夺秒，小成申请停掉可疑的定时任务，发现不见效，尝试业务流量切换到备用集群观察。

流量切换后，问题恢复了。

“切换到备用集群竟然可以恢复，难道是老集群的问题？”

想着定时任务还停着，小成建议申请恢复定时任务，数据同步定时任务再次被拉起。

这时已经过去了近两个小时。虽然切换新集群业务恢复，定时任务也恢复运行，但根本原因还是没查明。小成开始复盘整个过程，思考各个疑点。

为什么数据同步定时任务会大量报错？虽然报错，为什么切换到新集群会恢复？

这时电话又响起了："小成，备用集群也出事了！"

小成脑子快速闪过，"基本确定是定时任务的问题了"。

小成申请再次停止数据同步任务，经同意后，立即执行。执行后，业务开始逐渐正常。时间已经到了下午。

小成和团队人员紧急召开复盘会议，基于业务影响可能性的评估，决策对业务主链路进行隔离。当天下午紧急对业务链路进行隔离操作，并且在 1 个多小时内完成业务流量验证通过。

不知不觉一下午时间就这样过去了，小成理了理思路，决定把问题来龙去脉理清楚，于是发起了次日集中问题排查的会议邀约，参与人员涉及多方技术团队。

【DAY2】

通过对监控数据的梳理，小成发现当天机房的负载均衡服务器和弹性计算服务器有异常流量，且网络还有丢包情况。但是这流量的来源无法查明。另外 T 应用 hang 住的时候关系型数据库服务上的耗时急剧增加。

【DAY3～DAY4】

基于这一点，技术团队就此问题进一步深挖。通过复查所有相关 SQL 语句和查询数据量以及数据库数据量，试图找到病根，但是最终发现并没有异常。

那到底是什么变更导致数据库响应慢的？

技术人员再次调整思路，尝试在 SIT 环境 mock SQL 查询超时的情况，果然，重现了关系数据库服务查询耗时急剧增加的现象。并且 T 应用上出现大量 Close_wait 的连接，直到应用服务器上的资源耗尽，导致应用 hang 住。

【DAY5】

业务一切正常，但是 SQL 缓慢的原因还无法具体定位。

不过，业务涉及数据同步定时任务查询的 SQL 倒是还存在不少的优化空间，业务技术人员在当晚就通过索引优化的方式固定执行计划。

【DAY6】

第 6 天，小成邀请了数据库团队和公有云等技术专家团队参与到了问题排查中。

终于问题的根源逐渐水落石出，数据库和云服务器专家们就这个可疑的数据库服务响应慢的问题进行层层排查。

深入分析

（1）应用服务器单台网卡为什么有大量异常流量？

这是因为服务器主机的日志同步机制运行时，会每 5 分钟读取一次日志，从而引起异常流量。

（2）应用服务端为什么会大量 Close Wait?

Close Wait 产生的原因在于数据库服务端等待超时后，主动断开连接，如果应用服务端在数据库断开之后能够响应及时关闭，就不会积攒大量 Close Wait，但是应用服务端在当天问题发生时并未马上关闭无用连接，所以大量的 Close Wait 状态过多，导致主机资源耗尽。

当天排查也发现合作机构前置应用在网关关闭连接后，业务查询还未结束时，前置应用的网关连接不会关闭。这一事实佐证了上面这一点。

（3）应用服务端为什么会大量报错“open too many files”？

应用服务器可以 open 的文件数有上限，通常不会到达这个阈值。如果出现那可能就是有文件句柄没有及时释放。由于前面有大量处于 close_wait 的 tcp 连接，每个连接对应一个套接字（socket），也对应一个文件句柄，这是导致报错的主要原因。

（4）数据库服务的耗时是否和数据库主机的 CPU 利用率有关?

分析：

数据库主机的各个逻辑 CPU 在问题发生期间的监控数据都是正常，该因素排除。

监控佐证：

如图 3-1 所示是数据库实例采集到的 CPU 的平均利用率监控信息，从监控来看，并没有明显变化。

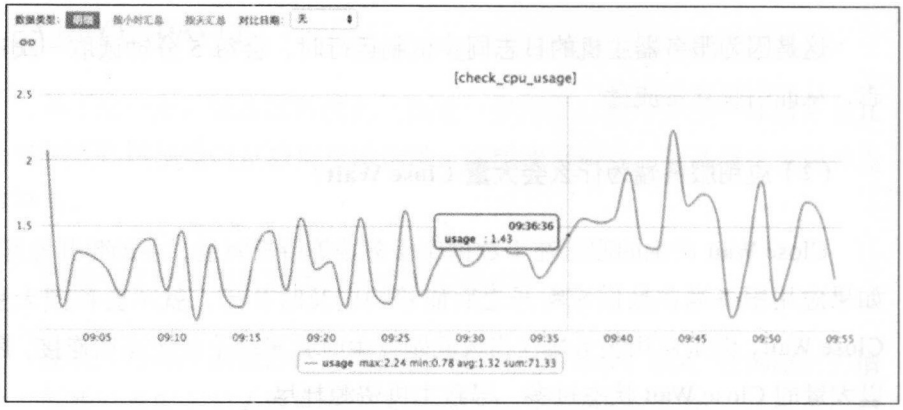


图 3-1 CPU 平均利用率

（5）会不会是数据库主机磁盘 IO 有问题？

分析：

磁盘 IO 监控包括每秒读写 IO 次数，每秒读写 IO 吞吐量、每个读写 IO 的平均响应时间，磁盘当前队列长度。

数据库服务的 DB 文件分为数据文件和日志文件，日志文件存放于 D 盘，数据文件存放于 E 盘，Tempdb 的文件在 D 盘。

模拟当天业务场景和大致时间点，从监控（如图 3-2 所示）可以看出，在 09：25 以前 D 盘的活动比较多，但总体都在正常范围。25 分之后，E 盘在 45 分左右有个高峰，但是峰值依然远低于磁盘的能力（最大 IOPS14000）。磁盘 IO 平均响应时间供参考。当磁盘 IO 压力过大或者出现访问问题时，那个响应时间会有值。正常情况下，数据都非常小（小于 1），总体来说数据库主机的 IO 负载也很低，此因素排除。

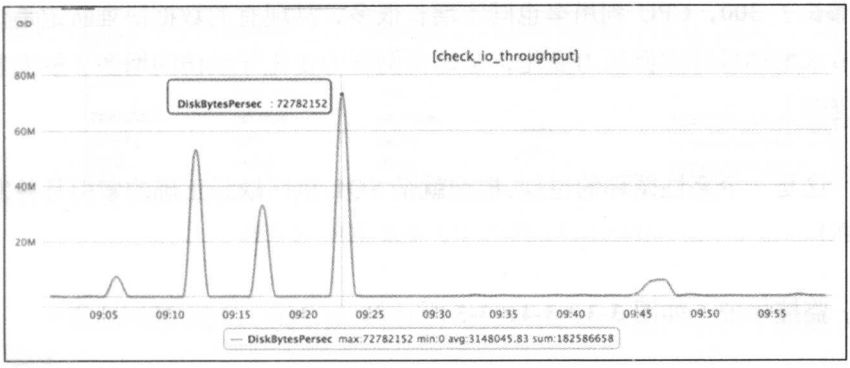


图 3-2　C 和 D 盘的每秒写吞吐量

（6）是不是和业务 SQL 性能有关 ？

分析：

关于 SQL 变慢，通过分析数据库日志发现，拉取同步数据的关键 SQL 缺乏正确的索引，进而导致其执行计划不是最优，理论分析这个执行计划也是不稳定的。随着业务同步数据的全量替换，以及表统计信息短时间内不能及时更新，SQLserver 可能会根据查询参数比对统计信息而调整执行计划。

推测在 09:35 左右，这个 SQL 的执行计划发生变化导致性能出现更差的情形，而高并发的访问加剧了这个性能的问题。这一点表现在 DB 上就是 CPU 利用率升高，CPU 利用率变高，反过来会影响 SQL 的性能（针对全体 SQL）。

09:37 前后相邻的两个诊断报告的等待事件表明 Buffer Latch 最大等待延迟也增加 180ms（这表示 Buffer Latch 等待更严重）。调用端观察结果就是 SQL 执行时间变长的 SQL 比例越来越多。

关于 SQL 变慢，还需要关注的是，相对于问题发生前一天，数据库连

接增长了 300，CPU 利用率也同比增长很多，从现有的数据很难断定是访问请求增加导致实例压力变化，还是实例压力变化导致访问增多（触发重试逻辑）。

这是一个恶性循环的过程，根源就是 SQL 执行缺乏正确的索引且并发很高！

监控佐证（如图 **3-3**、**3-4**、**3-5** 所示）。

SQL 耗时分析报告，如表 3-1 所示。

表 3-1 SQL 耗时分析

时间段	SQL 执行成本(平均逻辑读)
09:25:00～09:37:18	2106
09:37:18～11:30:18	203261
11:30:18～13:26:18	813672

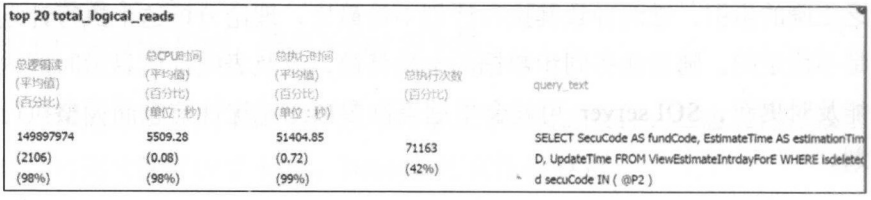


图 3-3 SQL 报告（09:25:00～09:37:18）

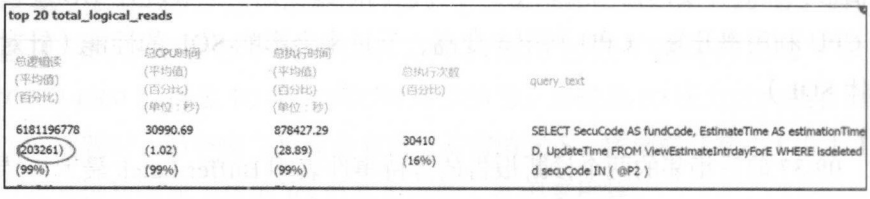


图 3-4 SQL 报告 09:37:18～11:30:18）

总逻辑读 (平均值) (百分比)	总CPU时间 (平均值) (百分比) (单位：秒)	总执行时间 (平均值) (百分比) (单位：秒)	总执行次数 (百分比)	query_text
5388136879 (813672) (99%)	18552.56 (2.8) (99%)	391670.86 (59.15) (99%)	6622 (7%)	SELECT SecuCode AS fundC D, UpdateTime FROM ViewE d secuCode IN (@P2)

图 3-5　SQL 报告（11:30:18～13:26:18）

重点问题 SQL，SQL 请求很高，且没有走上合适的索引（如图 3-6 所示）。

```
1  SELECT SecuCode AS fundCode, EstimateTime AS estimationTime, EstimatePercent AS esti
2  mationGrowthRate, estimateNetValue, TimeID, UpdateTime
   FROM ViewEstimateIntrdayForE WHERE isdeleted = 0 AND EstimateTime BETWEEN @P0 and DA
   TEADD(mi, 1, @P1 ) and secuCode IN ( @P2 )
```

图 3-6　问题 SQL

问题当日 09:02～09:37 期间 总执 次数:71163 次（实际从 09:25 开始），平均消耗时间 0.72s。

问题当日 09:31～09:56 期间 总执 次数:22914 次，平均消耗时间 16.0s。

分析：

视图 ViewEstimateIntrdayForE 的定义（如图 3-7 所示）。

```
select a.SecuCode, b.EstimateTime, b.TimeID, b.EstimatePercent,
b.EstimateNetValue, b.UpdateTime,b.IsDeleted, b.dcmguid as guid
from FundData.dbo.SecuCodeRelationSMCode a
inner join EstimateIntraday b on a.DianchaoCode = b.Code
where TimeID between 0 and 240
```

图 3-7　视图定义

表 FundData.dbo.SecuCodeRelationSMCode 的索引（如图 3-8 所示）。

Index_name	Index_description	Index_keys	dba备注
idx_dianchaoCode	noclustered located on Primary	DianchaoCode	这是原有的索引
SecuCodeRelationSMCode_ind2	noclustered located on Primary	SecuCode, DianchaoCode	这是原来没有的，在[illegible]加上的正确索引

图 3-8 SQL 索引情况

小结

本文这个案例的发生，一层层剥开来看，最后还是 SQL 执行计划不佳引起。因为缺乏合适的索引，所以原 SQL 的执行计划很可能处于不稳定的状态，即 SQLServer 可能会根据压力状况、数据量、统计信息以及传参数值来判断当前的执行计划是否为最优，如果数据库基于内存中的信息判断有更好的选择（从数据库角度判断，这很可能是个更加错误的选择），就改变当前执行计划，从而导致性能上可能出现更大的延时，再加上高并发，就进一步放大了这一性能问题。

基于本次问题的整个过程处理和原因分析，可以总结出以下几点措施和优化点来避免类似问题重复发生：

（1）如何避免服务器异常流量？

优化服务器主机日志同步方式，可考虑迁移现有成熟的日志同步工具。

（2）大量 close wait 的异常怎么才能避免？

从业务上下游主链路梳理超时时间参数和 SQL 超时设置，对齐超时标准，统一调优。

（3）如何避免“open too many files”？

检查文件读写操作、socket通讯是否正常。

（4）业务如何避免同类SQL执行计划问题发生？

针对TOP10 SQL集中review执行计划和索引设置，避免触发SQLServer的可能的潜在问题。

数据库主机虽有CPU等系统监控，但是并无执行计划的监控，完善SQL执行计划的监控覆盖。

（5）线上问题迅速发现

完善依托公有云的业务应用监控，尤其是远程调用和数据库访问的监控数据完善，便于后续第一时间发现问题。

完善添加关系型数据库SQL执行计划监控项，添加数据库容量带宽和连接数的监控。

（6）问题快速定位

线上问题快速定位和排查能力提升，定位问题需要分阶段定位，首先定位外部系统原因，数据库原因，网络原因还是应用程序原因。本次在定位是数据库的原因耗时太长，丧失了数据库层面的第一现场。

问题处理环节，需以最快的时间协同不同团队之间的响应处理，避免错失问题排查的最佳时间窗口。

3.2 波谲云诡，数据库延迟

背景

某业务在生产环境遇到了一个诡异的数据库访问现象：应用通过数据库中间件 TDDL 访问 MySQL 数据库，偶尔会出现超长的访问延迟，导致整个远程服务接口调用超时。从调用链分析平台 EagleEye（分布式调用的跟踪者）看到的图形是这个样子（如图 3-9 所示）。

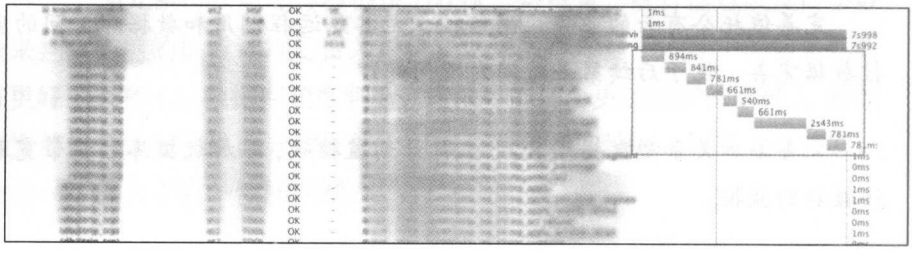


图 3-9 EagleEye 调用链监控

从这张图上可以看出：

（1）访问多张表的延迟都超过 500ms；

（2）同一笔调用，查询另一些表的延迟只有 1ms；

（3）都是访问同一台数据库。

在什么情况下会出现这种现象呢？发现下列 3 种情况，问题完全难以捉摸：

（1）大约0.3%的服务调用出现这种超时，其他请求完全不受影响；

（2）几乎不受业务访问压力的影响，包括预发环境也能重现这种超时；

（3）所有应用服务器上都能重现。

问题分析

会不会是应用、中间件、网络、数据库上产生了资源等待？

答案是不会。因为所有业务请求都需要相同的资源，大家都在同一个资源池上排队。假设请求是相同的，排队是公平的——如果有人排队等了500ms才拿到资源，有人1ms就拿到了资源——那么，一定会有等待了100ms，200ms，300ms，400ms这样的请求，访问延迟应该体现平均分布。

但现象与预设显然不符。那什么场景下才会产生明显的延迟差距？有以下几种情况：

（1）锁等待

有几行数据被加锁，造成部分请求长时间等待锁。或者，出现非公平的锁等待，插队太多，请求被“饿死”。

（2）热点数据

部分业务ID返回的数据量很大，或者需要扫描的索引数据行数很多。

（3）IO抖动

高IO压力下，命中缓存的请求和需要访问磁盘的请求延迟（因为IO排队）差别会很大。

（4）未知问题

除以上外，也不能排除是应用、中间件、或者数据库的未知问题导致。

首先，锁等待和 IO 抖动的问题被排除了，因为发生问题的数据库请求都是 SELECT 语句，MySQL InnoDB 引擎支持多版本并发控制（MVCC），普通 SELECT 查询是不加锁的。

此外，数据库压力很低，缓存命中率很高，可以排除是 IO 的原因（如图 3-10 所示）。数据不均匀的可能性也被排除了。业务查询中有一部分是主键查询，符合条件的数据有且只有一条。并且，仔细筛查 EagleEye 平台提供的调用链记录发现：相同参数的远程服务调用，也会出现有时快、有时慢的现象。

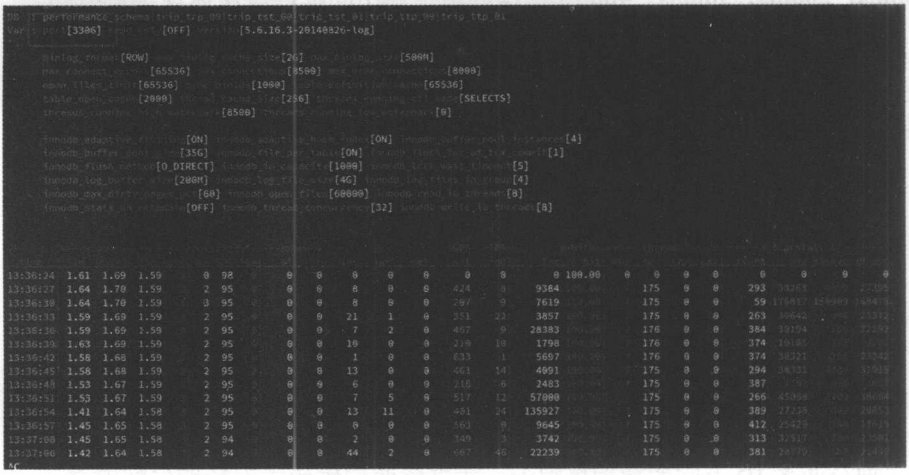


图 3-10　应用数据库监控

最后，DBA 在业务数据库监控上完全看不到慢查询。上文 1、2、3 三种情况导致的 SQL 执行时间大于 1s 都会记录在 MySQL 慢查询日志。从 EagleEye 平台可以看到最高延迟超过 2s，但是 MySQL 数据库的慢查询日志却没有任何记录。

已知可能性都被排除了，所以只能通过日志分析寻找问题特征。

日志分析

从数据库中间件 TDDL 记录的统计日志 tddl-stat.log 发现：业务数据库使用了分库分表，一共包含 8 个分库，高延迟查询都来自其中的两个分库。

```
V1 201x-xx-xx 16:24:24:629 read statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 times:1779 errors:0 qps:29
rt(avg/min/max):94/0/2164 ]
V1 201x-xx-xx 16:24:24:630 write statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 times:6 errors:0 qps:1
rt(avg/min/max):0/0/1 ]
V1 201x-xx-xx 16:24:24:630 dataSource statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 poolSize:30
active(now/max):3/9 pooling(now/max):10/13 create:0 destroy:0 errors:0 ]
V1 201x-xx-xx 16:24:24:630 read statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 times:1647 errors:0 qps:27
rt(avg/min/max):69/0/2164 ]
V1 201x-xx-xx 16:24:24:630 write statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 times:14 errors:0 qps:1
rt(avg/min/max):0/0/1 ]
V1 201x-xx-xx 16:24:24:630 dataSource statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 poolSize:30
active(now/max):2/8 pooling(now/max):11/13 create:0 destroy:0 errors:0 ]
V1 201x-xx-xx 16:24:24:631 read statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_02_3307 times:1829 errors:0 qps:30
rt(avg/min/max):0/0/1 ]
V1 201x-xx-xx 16:24:24:631 write statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_02_3307 times:10 errors:0 qps:1
rt(avg/min/max):0/0/1 ]
V1 201x-xx-xx 16:24:24:631 dataSource statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_02_3307 poolSize:30
active(now/max):0/8 pooling(now/max):8/8 create:0 destroy:0 errors:0 ]
...
```

可以看出 00 和 01 库的查询延迟指标：rt(avg/min/max):94/0/2164 与下面的 02 库 rt(avg/min/max):0/0/1 完全不同，而且每个周期的统计日志重复该现象。

巧合的是，00/01 库正巧位于同一个 MySQL 实例 10.x.77.4:3306 上，业务的其他分库则全部位于同一台物理机的其他 MySQL 实例（10.x.77.4:3307~3309）上。这说明问题更可能与这个数据库实例有关。因为分库分表后，数据库中间件 TDDL 处理不同分库没有区别，不会导致 00/01 库出现高延迟查询，而 02-07 库完全没有问题。

DBA 检查了 10.x.77.4 上的 4 个 MySQL 实例，没有发现 3306 端口的实例配置有特殊差异。从数据库监控看，3306 与其他实例的监控曲线也没有明显的差别。

由于应用端统计到的数据库延迟很高，而数据库端看到的查询执行很快。怀疑的目标转移到了网络上。难道是网络问题？抓个包看看（如图 3-11 所示）。

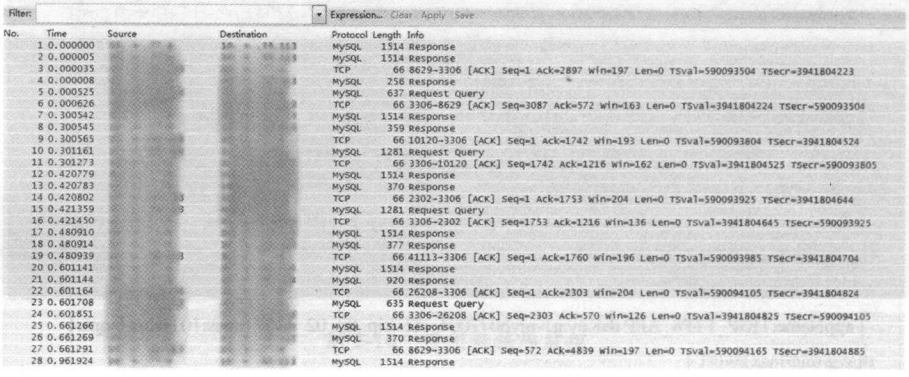


图 3-11 数据库抓包分析-1

抓包分析

```
tcpdump host 10.x.77.4 and port 3306 -s 0 -w dump.pcap
```

Wireshark 提供的 RTT（Round Trip Time）统计表明网络延迟很低（如图 3-12 所示）。

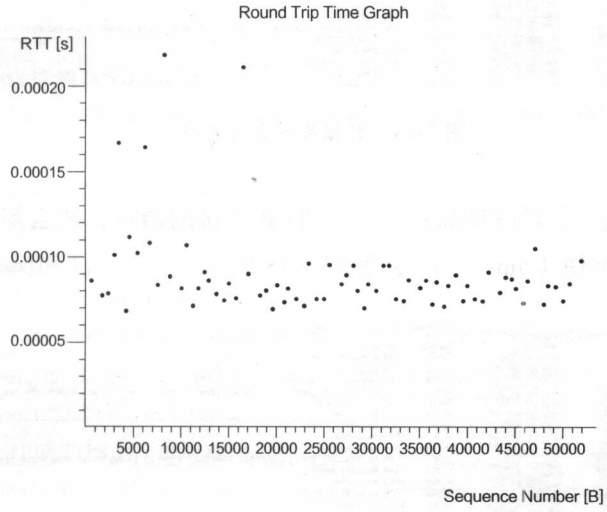


图 3-12　网络延迟统计

如图 3-13 所示，从抓包的确可以看出有一些 MySQL 请求的延迟很高（600ms）：

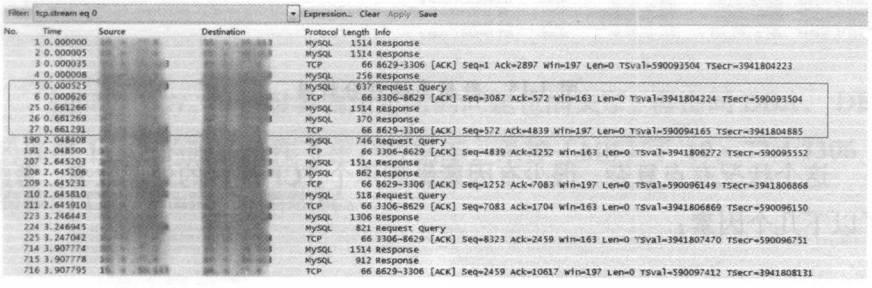


图 3-13　数据库抓包分析-2

如图 3-14 所示，大多数请求的延迟比较正常（3ms）。

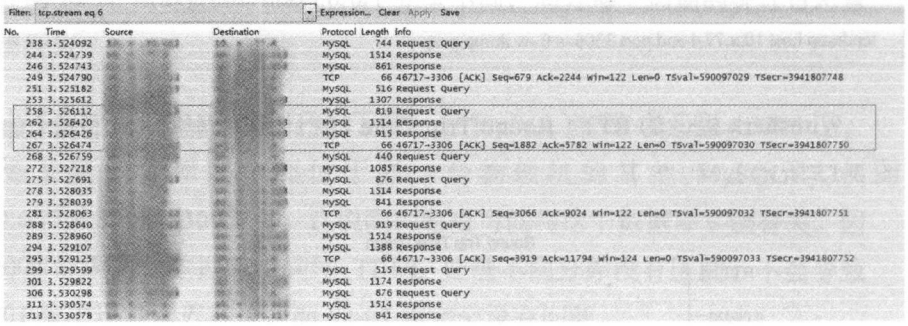


图 3-14　数据库抓包分析-3

但是发现一个特殊现象：同一条 TCP 连接的请求，要么都慢（500ms 以上），要么都快（5ms 以下），如图 3-15 所示。

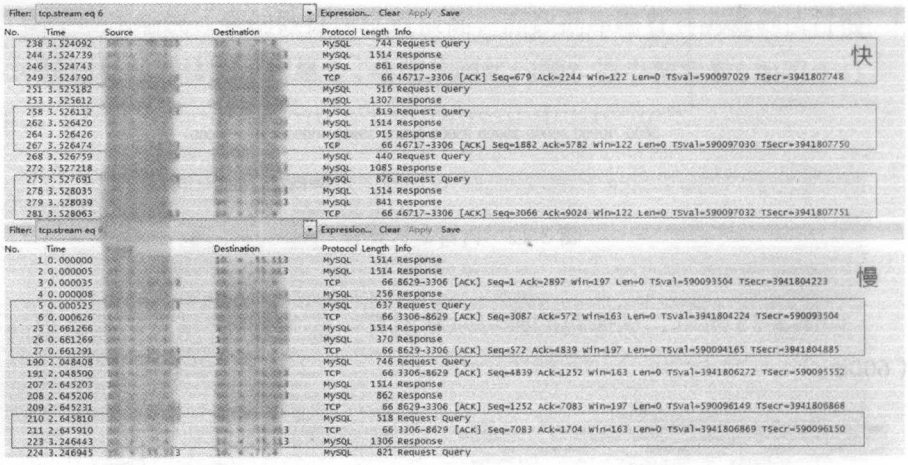


图 3-15　数据库抓包分析-4

这个状况有点复杂，很少有因素影响单个 TCP 连接的处理效率。考虑了以下几个因素：

- 网卡硬件问题？

- TCP 连接分配的网卡中断处理不均衡？
- 负责处理请求的 MySQL 线程被绑定到繁忙的 CPU 上？

结果都不是。

抓包后继续监控 TDDL 统计日志，发现高延迟查询的现象从 16:42 起完全消失了，现在 00/01 库的延迟统计数据与其他分库一样稳定：

```
V1 201x-xx-xx 17:20:24:629 read statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 times:1826 errors:0 qps:30
rt(avg/min/max):0/0/6 ]
V1 201x-xx-xx 17:20:24:629 write statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 times:24 errors:0 qps:1
rt(avg/min/max):0/0/1 ]
V1 201x-xx-xx 17:20:24:630 dataSource statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_00_3306 poolSize:30
active(now/max):0/7 pooling(now/max):11/11 create:0 destroy:0 errors:0 ]
V1 201x-xx-xx 17:20:24:630 read statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 times:2393 errors:0 qps:39
rt(avg/min/max):0/0/3 ]
V1 201x-xx-xx 17:20:24:630 write statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 times:13 errors:0 qps:1
rt(avg/min/max):0/0/3 ]
V1 201x-xx-xx 17:20:24:630 dataSource statistics
 [ appName:TRIP_TTP8_APP dsKey:g1-myb077004_et2_trip_ttp_01_3306 poolSize:30
active(now/max):0/7 pooling(now/max):13/13 create:0 destroy:0 errors:0 ]
.....
```

询问业务方，得知服务调用超时的问题也消失了。再询问 DBA，DBA 的确在 16:42 的时候操作了生产数据库：杀掉（kill）了 10.x.77.4:3306 上的多余 DRC Binlog 采集线程。

什么是 DRC？

DRC 是一个阿里集团内部采集数据库增量数据的服务。它使用 MySQL 主备复制协议，在数据库启动一个长时间运行的 Binlog 采集线程，持续抓取新生成的 MySQL Binlog 日志传输到另一端的 DRC 服务节点。

为什么杀掉 DRC Binlog 采集线程会改变业务的访问延迟？之前在 MySQL 上从没有遇到过。因为 DRC 使用 MySQL 内置的主备复制协议，对数据库的影响理论上跟一台 MySQL 备库没有区别。 尽管 DRC 会长期占用一个 MySQL 线程，但是普通 MySQL 是一个线程服务一个连接的调度方式，除了 IO，锁和 CPU 切换，不同连接上的请求不会相互影响。

咨询 DBA 才知道：业务的 MySQL 最近从 5.5 升级到 5.6，新增了线程池功能，问题的核心出在线程池。

MySQL 线程池

MySQL 线程池是个高大上的功能。拥有线程池的 MySQL Percona 分支以及官方 MySQL 企业版声称可以“提升 MySQL 扩展性 60 倍”。它的优势是能让 MySQL 用较少的线程支持更多的数据库连接与网络并发。因为线程数太多会在锁和 IO 上产生更多的竞争，使得更多的 CPU 资源消耗在上下文切换、自旋（spin）和内核调度上。

MySQL 的线程池相比类似组件做了更多的优化。其中一个优化是对线程资源进行了分组，每个分组只有少量（2～3 个）线程。这样每个组的资源相互隔离，防止慢查询耗尽全部线程资源，导致 MySQL 无法响应业务请求。

在分组模式下，如果有新的 TCP 连接上 MySQL，新连接会以散列的方式固定分配到一个分组，由这个分组内的线程负责响应连接上的请求。当处理结束，响应请求的线程释放连接，接着去处理同一个分组内其他 TCP

连接上的请求。

但是上面已经提过，DRC 发起的连接与普通连接不一样。它请求的 Binlog Dump 任务始终不会结束，这样执行 DRC 请求的线程就会一直无法释放。而 MySQL 分组内的线程实在太少了。如果正好有一个分组分配到 2 个以上的 DRC 连接，该分组内就没有空闲线程来响应其他请求了。因此，被固定分配到这个分组的 TCP 连接得不到线程处理，表现在应用端就是高延迟。

注：这里表现出高延迟而不是超时，原因是 MySQL 线程池有另外一个参数 thread_pool_stall_limit 避免无限期等待。请求如果在分组内等待超过 thread_pool_stall_limit 时间没有被处理，则会退回传统模式，创建 MySQL 线程来处理请求。这个参数的默认值正好是 500ms。

顺便由于请求是堆积在 TCP 缓冲区中没有被处理，MySQL 端是无法记录到这类请求的等待时间的。这样，MySQL 的慢查询日志就不会有记录。

知道原因后，解决问题就简单了：

- 临时解决方案是尽量减少单个 MySQL 实例上的 DRC 连接的数目。
- 最终解决方案是优化 MySQL 线程池，让 Binlog Dump 这类特殊请求归属到独立的线程池处理，或者绕过线程池，回归到一个线程服务一个连接的模式。

小结

本文这个案例的分析和排查过程比较有意思。其实一开始的分析已经接近了问题的本质：资源池的均衡问题——如果所有请求都在同一组资源上排队等待，那么访问延迟一定是连续分布的。MySQL 的线程池分组功能实际上拆分了资源池，使得不同的请求分别在不同的资源池上排队。当这

些池中的资源数量不同，那么就一定会有请求更容易得到资源，有些请求拿不到资源，出现类似的延迟分布。

另外，从排查过程可以发现，EagleEye 这样的调用链分析平台对判断分布式系统问题非常关键，利用平台提供的调用链数据，可以在初期迅速排除掉一系列可能性，将问题聚焦。在线上问题处理中，排查人员往往面临快速定位原因，恢复系统的压力。这时，聚焦问题就是最大的帮助。

3.3 风暴来袭，AliSQL 连接池调优

某个业务数据库在凌晨进行压测，压力比较大的 4 个节点 DB 出现瞬间 QPS 跌 0，并出现了大量"" "unauthenticated user"。AliSQL 的 session 状态如下：

```
13716534 ctu ××.××.×××.×××:××××× sec_credibledb Sleep 1222 NULL
13716535 ctu ××.××.×××.×××:××××× sec_credibledb Sleep 1222 NULL
   * 12
15922528 unauthenticated user ××.××.×××.×××:××××× NULL Connect
 NULL Reading from net NULL
15922535 unauthenticated user ××.××.××.×:××××× NULL Connect NULL
 Reading from net NULL
   * 347
   15923418 unauthenticated user connecting host NULL Connect NULL login
NULL
   15923419 unauthenticated user connecting host NULL Connect NULL login
NULL
   *3265
```

原因分析

当天，业务开发、中间件开发和 DBA 一起，深入分析了可能的原因。

MySQL 的登录过程如下：

（1）Client 建立 TCP 连接到 DB 对应端口。

（2）DB 向 Client 发送握手包。

（3）Client 向 DB 发送验证信息包（用户名/密码等）。

（4）DB 向 Client 发送验证确认包。

从网络包的层面来看是如下的过程（如图 3-16 所示）。

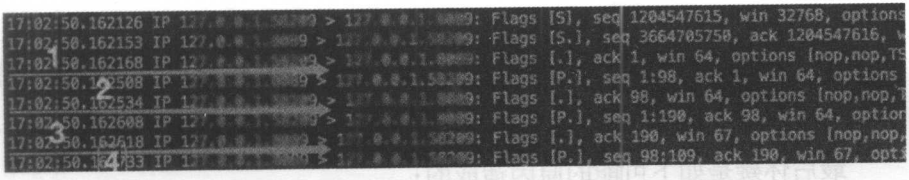


图 3-16　MySQL 登录的网络交互过程

其中 127.xx.xx.xx:58209 是 client，127.xx.xx.xx:8089 是 server（DB）。

初步分析

首先排除了 skip-name-resolve 的错误。

其次排除了是由于数据库处理登录性能过慢导致拥堵（如图 3-17 所示）。

图 3-17 出现拥堵是的 AliSQL 线程状态

可以从 mysqld 的 backtrace 中看到，此时有大量的线程正在等待应用发送密码包（mysqld 已经发送了握手包—步骤 2，正在等待密码包—步骤 3。但因为 MySQL 的发包是异步 IO，所以其实并不能确定步骤 2 的 Client 到 DB 的回包有没有被收到。

同时从 backtrace 中看到了 DB 非常的空闲，几乎没有正在执行的 SQL。这也和 show processlist 的结果互相印证。

最后怀疑是如下可能的原因造成的：

（1）DB 端 OS 层调度问题，导致握手包无法被发送 or 密码包无法被接受—小概率。

（2）Client 端（应用），因为 OS 层 or 本身调度繁忙导致握手包无法被接收 or 密码包无法被发送—小概率。

（3）网络问题导致握手包 or 密码包丢失。

（4）Client/DB 端由于其他原因导致和握手协议不一致，引发握手失败。

验证推理

通过压测，技术人员分别在 Client 和 DB 端进行了抓包，复现交互过

程，具体定位是哪种情况导致（如图 3-18 所示）。

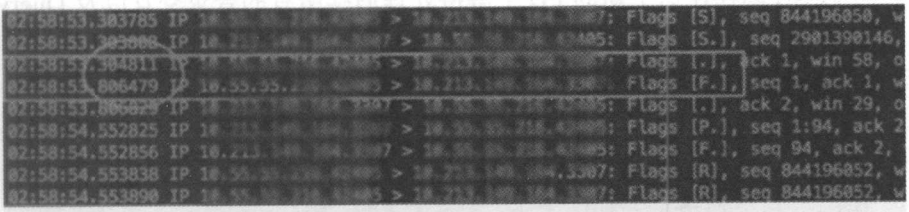


图 3-18　现场 1

首先看到了这种场景，在 Client 和 DB 建立 TCP 连接完成后大概 0.5 秒 Client 端主动关闭了 TCP 连接。但是 DB 在 0.7 秒之后才发送握手包，当 DB 发送握手回包的时候，其实 TCP 连接已经关闭，因此发送失败，DB 也确认关闭 TCP 连接。首先需要确认 Client 端主动关闭连接的原因。

在上面的 MySQL 登录交互中，对于 Client 来说在步骤 1 建立 TCP 连接和步骤 2 接收握手包之间，是需要进行等待的，这个等待必然有一个超时，在 JDBC 中这个超时就是 connectTimeout。

这个参数默认是 0，也就是无超时等待，排查确认应用中对这个超时的设置为 500ms。

```
<!- - 建立连接的超时时间，毫秒为单位 - ->
<entry key ="connectTimeout" value="500" />
```

而 DB 端同样存在这样的超时（在步骤 2 发送完成握手包和步骤 3 接收密码包之间），这个超时是 connect_timeout，超时时间默认是 10s，设置的是 8s。

其次要确认下 DB 端为什么会超过 500ms 才发送握手包。

这里先说明下线程池和 MySQL 的登录请求处理：

步骤 1 在 MySQL 引入线程池之前，在 TCP 连接建立后，DB 会新建一个线程，单独在这个新的 FD 上等待处理后续所有的登录交互以及 Query 交互过程。

步骤 2 在 MySQL 引入线程池以后，在 TCP 链接建立后，主线程会根据 thread id 将 FD 绑定到对应的 group 的 EPOLL 中，有 EPOLL 来监听后续的网络包。如果发现这个 session 尚未登录，通用的 worker 会接收处理登录请求（在线程池中一次完整的登录交互，相当于一次 SQL 查询交互）。

但是线程池中的 worker 数量是有限的，根据应用的设置，可用的 worker 大概在 320 到 354 之间（部分可能会作为 listener 存在）。

在第一部分的 show processlist 的结果中技术人员知道了有 347 个 session 处于如下状态：

```
15922528  unauthenticated user 10.XX.XX.XX:59856  NULL  Connect NULL
Reading from net  NULL
```

这个状态的 session 正处于步骤 2 和步骤 3 之间，在等待 Client 发送密码包。这个状态最多会维持 8s（connect_timeout），这也说明了 Client 存在不发送密码包也不关闭 TCP 连接的 CASE（如果 Client 关闭 TCP 连接，EPOLL 会被触发，并引起等待终止）。

同时有 3265 个 session 处于如下状态：

```
15923418  unauthenticated user  connecting host NULL  Connect NULL
login  NULL
```

这个状态是主线程把建立 TCP 连接完成的 session 放入线程池队列中，正在队列中被等待调度的状态，但是因为已有的所有 347 个可用 worker 被其他登录任务所占据。因此没空闲的 worker 来处理这些请求；而这 347 个 worker 必须等待 8s 的 connect_timeout 超时后才会退出，因此导致其他 3265

个连接必须等待一定的时间以后，才能发送握手包给 Client（上文的 case 中等待了 0.5+0.7=1.2s），但此时 Client 已经把连接关闭，因此握手包发送失败，继而导致登录失败。

接着找出了 Client 没有关闭 TCP 连接的 CASE（如图 3-19 所示）。

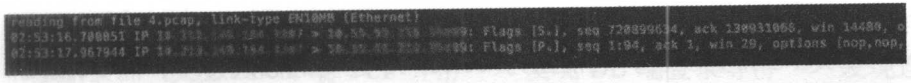


图 3-19　现场 2

上面这个案例中，技术人员发现在步骤 1 TCP 连接建立后的 1.26 秒以后，DB 端发送了握手包。期间 Client 端并没有断开 TCP 连接，而且在 DB 发送握手包以后，Client 也没有返回 ACK or 密码包。

这种 case 下，会引发 DB 端线程池空等 8s，并最终握手失败，导致线程池 worker 被耗尽。

但是 Client 端在 0.5s 的等待后，会再次发送新的链接请求，进入线程池的等待队列，引起了连接池的雪崩。

技术人员检查了业务的 JDBC 版本，发现为 mysql-connector 5.0.5 版本，这个是 2007 年 3 月 2 日发布的一个版本，其存在和现有的 MySQL 版本的兼容性问题，可能导致 TCP 连接未异常关闭而退出（怀疑是由于版本过老，这不再深究代码）；而同样访问业务 DB 的另一个业务使用 mysql-connector 5.1.3X 版本则不存在这个问题。

到此确认了 DB 出现长时间“unauthenticated user”的原因。

最后需要确认最初引起 Client 端 connectTimeout 的 0.5s 超时的原因（如图 3-20 所示）。

```
Last packet sent to the server was 0 ms ago.
        at com.mysql.jdbc.MysqlIO.readPacket(MysqlIO.java:62
        at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:1
        at com.mysql.jdbc.Connection.createNewIO(Connection.
        at com.mysql.jdbc.Connection.<init>(Connection.java:
        at com.mysql.jdbc.NonRegisteringDriver.connect(NonRe
        at com.alipay.zdal.datasource.resource.adapter.jdbc.
```

图 3-20 Client 的报错信息

业务日志排查发现，之前因为 select X 之类的心跳过多，而关闭了 Client 连接存活维持机制，导致了连接池中出现了大量的死链接，在压测流量开始的时候，这些死链接一起发起重连，形成并发数在 2000 以上的连接风暴。

如此压力的连接风暴下，线程池势必会出现超过 0.5s 的握手延迟，继而引起了 Client 端的 connectTimeout 超时重连，最终导致连接雪崩。

解决方案

确认了产生这次问题的所有原因后，技术人员讨论出了如下结论：

（1）确保连接重置一定被发送，避免 Mysqld 无谓等待（先升级 jdbc 版本尝试）；

（2）Client 连接池保持存活机制，避免连接风暴；

（3）Mysqld 增加 worker 数量，提高被连接风暴冲击能力（存在副作用）；

（4）Client 增加 JDBC 的 connectTimeout 参数，避免无谓的重连；

（5）Mysqld 减少 connect_timeout 时间，避免无谓的等待（最小间隔只能是 2s，效果不大）。

最后技术人员升级了 JDBC 版本，添加了保持存活机制（autoReconnect），修改 Client 端超时参数 connectTimeout 到 1s，并最终通过验证解决了这个问题。

完整的过程如下：

发现问题 => 排除解析原因 => 排除 DB 负载问题 => 发现 Client 端超时 => 发现 Client 端 TCP 异常 => 发现 DB 端登录等待 => 发现连接风暴原因 =>最终确认解决问题

小结

本次连接风暴异常时的现象类似于 DoS/DDoS 中的 SYN Flood 攻击。其原理都是一样的：Client 在完成第一步握手后（TCP 中是数据包，MySQL Protocol 是建立 TCP 连接），再也没有后续的行为（TCP 中是 ACK，MySQL Protocol 是发送密码包）。当有大量类似连接的时候，服务端都需要维护大量的半连接，导致服务端消耗了大量的资源，影响了正常的服务请求。

MySQL Protocol 在遇到类似行为的时候，会更加的严重，因为 MySQL Protocol 的上下文相对于 TCP 会更重一些。常规配置下的 MySQL 大约只能承受 100 以下 connection per second 的 SYN Flood 攻击。这会使得暴露端口的 MySQL 变得非常的脆弱。AliSQL 针对这个安全薄弱点，整体重构了 MySQL 的握手验证的服务端实现，使得 SYN Flood 的防御能力提升到 3 个数量级以上，极大地增强了 AliSQL 的安全性。

3.4 防患于未然，ORM 规约变更案例

背景

Select * 带来的便利是对于字段较多的情况下，可以少写字段，不会出现在业务代码中遗漏字段的情况。但大家应该也了解，select *会查询出数据库的所有字段，如果某个字段很大，可能对性能有影响，一般建议按需使用字段。如果认为 select *的弊端仅仅是以上这些，那就大错特错了。为什么 DBA 强烈禁止使用 select *，阿里巴巴集团开发规约也把其作为禁止使用的条例，肯定是有原因的。

在某次项目发布阶段，因为业务需要新增表字段，从 SQL 的代码逻辑来看，使用了 select *，新增字段应该是兼容的，但在做线上数据库 DDL（Data Definition Language，数据库模式定义语言）操作之后，立即出现了日志错误数飙升报警，说明数据库新增字段并不兼容。

问题排查过程

（1）查看错误日志，发现是数据库查询报错，数据库中缺少字段。

```
  org.springframework.jdbc.UncategorizedSQLException: SqlMapClient
operation; uncategorized SQLException for SQL []; SQL state [null]; error code
[0];   --- The error occurred while applying a parameter map.   --- Check the
MS-FIND-RECEIVEADDRESS-BY-ID-USERID-InlineParameterMap.   --- Check the
statement (query failed).   --- Cause: Unknown column
'RECEIVE_ADDRESS.town_code' in 'field list'   at
sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)   at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAc
```

```
cessorImpl.java:57)  at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConst
ructorAccessorImpl.java:45)  at
java.lang.reflect.Constructor.newInstance(Constructor.java:526)  at
com.mysql.jdbc.Util.handleNewInstance(Util.java:389)  at
com.mysql.jdbc.Util.getInstance(Util.java:372)  at
com.mysql.jdbc.SQLError.createSQLException(SQLError.java:980)  at
com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3835)  at
com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3771)  at
com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2435)  at
com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2582)  at
com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2535)  at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:1
911)  at
com.mysql.jdbc.PreparedStatement.executeQuery(PreparedStatement.java:2034
)
```

（2）从日志来看，和数据库表结构变更的表名和出错字段都相同，可以确定问题就是表变更引起的。第一反应是新增字段，业务代码中字段映射出错，马上准备执行回滚。此次变更的 DDL 如下：

```
ALTER TABLE `receive_address`
ADD COLUMN `town_code` varchar(16) COMMENT '街道或镇编码'
```

（3）当回滚还未执行，日志错误就已经自动恢复，没有新的错误生成，于是开始进行问题排查。

（4）从问题的现象来看，这个问题只有在变更过程中才出现，不太像是结果集映射问题，如果是映射问题，不执行回滚是无法自动恢复的。DBA 反馈，可能是 TDDL（Taobao Distributed Data Layer 分布式数据访问引擎）层对 Select * 的解析逻辑引起 DDL 变更的不兼容。

（5）联系 TDDL 了解相关逻辑，确认的确会从分库分表的第一张表获取元数据信息，并做缓存（几分钟后失效），在目标库执行 SQL 时用的就是第一张表的结构字段，所以在做新增字段变更过程中可能会出现以上错误。

解决过程及原因

解决过程如下：

（1）“receive_address” 表(分库分表)新增了一个字段；

（2）SQL 语句存在 “select * from receive_address”；

（3）由于是分库分表，共 N 个库（N 大于 4），数据库表变更的时候，是分库分批执行的。N 个库变更完成，历时 M 分钟（时间长短取决于分库分表数等因素）；

（4）TDDL 在执行的时候，碰到 select *，会从数据库表中解析出对应的全部字段(取第一个库的第一个表进行解析，解析之后，会缓存结果)，替换*，然后再把解析后的 SQL 语句提交到目标数据库执行；

（5）在第一个库变更之后，TDDL 拿到最新的字段列表，后续一段时间内的查询，都直接用带有新增字段的 SQL 语句提交到数据库执行；由于有部分数据库还没执行变更，没有新的字段，导致数据库执行出错，无法查询数据。

该问题的原因为老代码中 select * 的使用，在分库分表数据库新增字段时，TDDL 内部没有兼容逻辑，导致 select * 被先解析成表 0 的字段，但在未变更完成的目标表中并没有对应新增字段，从而出现字段找不到的错误。具体流程如图 3-21 所示。

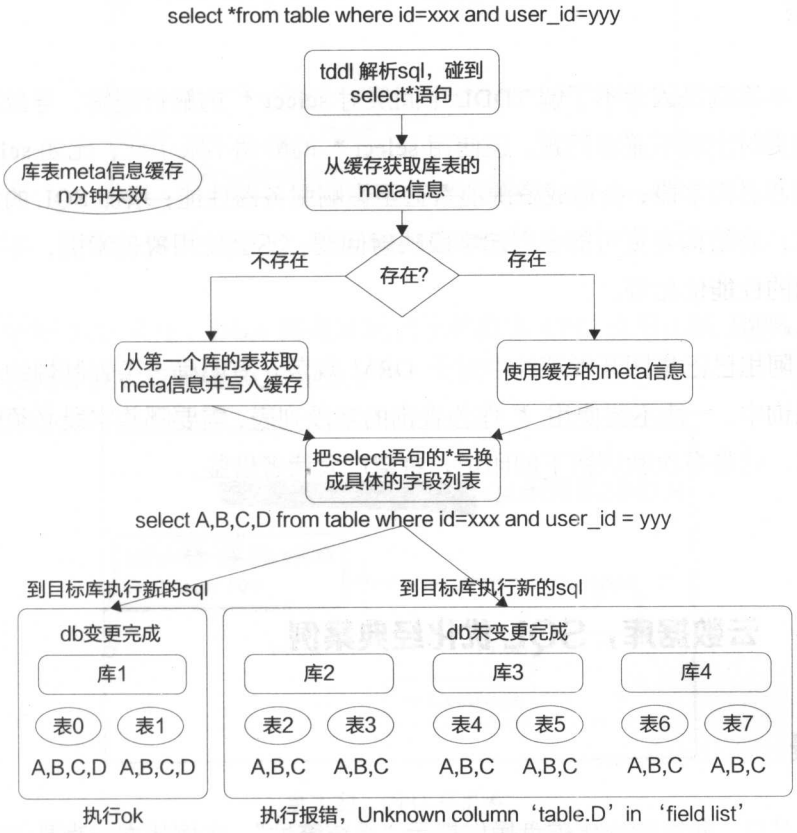


图 3-21　变更流程图示

对于此问题，只在分库分表数据库新增字段变更过程中有问题。但为了避免以后变更的不兼容，还是将所有 select * 的用法去除，改为具体字段，这样不管 TDDL 是否有兼容逻辑，都保证没有问题。

在 TDDL 中间件层面，也对原有的 select * 的解析逻辑进行优化，让系统的兼容性更强。思路是当 TDDL 内部检测到一次类似 SQLSQL 报错时，从目标库去获取数据表元数据信息做缓存。

总结

本案例是因为不了解 TDDL 中间层对 select * 的解析逻辑，导致数据库变更时出现不兼容问题。但使用 select * 的弊端不限于此，比如 select * 查询非必需字段，会造成资源浪费甚至影响服务器性能；增加 SQL 的解析成本；表结构变更可能会引起字段映射问题；不会使用覆盖索引，不利于查询的性能优化等。

阿里巴巴集团开发规约中对于 ORM 规范，有明确一条强制规约：在表查询中，一律不要使用 * 作为查询的字段列表，需要哪些字段必须明确写明。这是多次教训留下的印记，希望广大读者借鉴。

3.5 云数据库，SQL 优化经典案例

背景

某日，开发初始化磁盘操作提示“系统繁忙”，实例状态一直是“重置中”，通过排查磁盘挂载失败的请求日志，发现数据库连接异常了。

是不是新切换的系统有什么问题？

进一步排查发现工作流数据库上的连接已经达到上限，同时失败的任务都卡在进系统之前的数据库操作上。

为什么数据库连接会达到上限呢？当前时间点是一天的业务高峰期，所以第一时间查看了用户的访问流量，但是每秒请求量没有明显波动，排除是用户自身流量增加导致，所以怀疑是数据库问题导致，开始联系 DBA 排查数据库问题（如图 3-22 所示）。

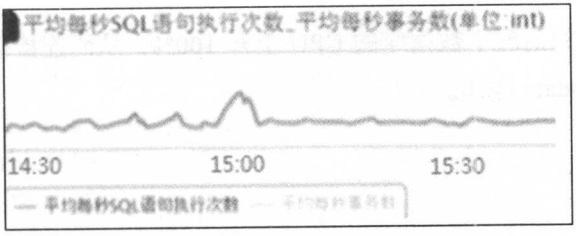


图 3-22　平均每秒 SQL 执行次数

如图 3-23 所示，DBA 排查发现当前数据库 CPU 使用达到 100%，怀疑存在慢 SQL，查看慢 SQL 记录。

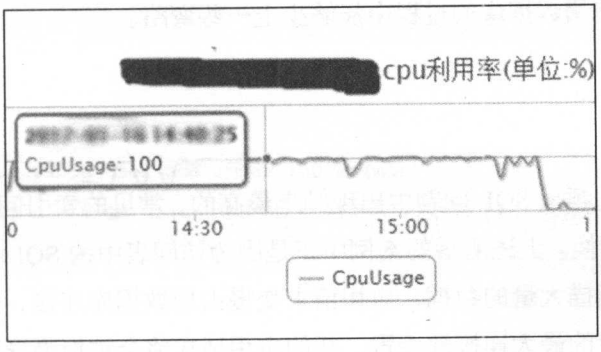


图 3-23　CPU 利用率

在慢 SQL 记录中发现一条慢 SQL 大量堆积，而且单条慢 SQL 执行时间均超过 1s（如图 3-24 所示）。

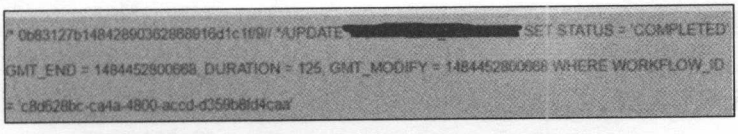


图 3-24　慢 SQL 记录

进一步排查确认是 WORKFLOW_XXX 表上没有加上 WORKFLOW_ID 的索引，也正是这条未加索引的慢 sqlSQL 导致业务高峰

时段高并发的情况下数据库的 CPU 上升 100%，从而连接数大量堆积导致无法执行 update 操作。

案例扩展

数据库问题中，由于 SQL 问题导致的数据库故障层出不穷，下面将遇到的 SQL 问题总结归类，还原问题原貌，给出分析和解决问题的思路，帮助读者在使用数据库的过程中能够少走一些弯路。

索引篇

索引问题是 SQL 问题中出现频率最高的，常见的索引问题包括：无索引，隐式转换。上述所举的案例中正是因为访问表中的 SQL 无索引导致全表扫描，扫描大量的数据，应用请求变慢占用数据库连接，连接堆积很快达到数据库的最大连接数设置，新的应用请求将会被拒绝导致故障发生。隐式转换是指 SQL 查询条件中的传入值与对应字段的数据定义不一致导致索引无法使用。常见隐式转换如字段的表结构定义为字符类型，但 SQL 传入值为数字；或者是字段定义 collation 为区分大小写，在多表关联的场景下另外的关联字段却不区分大小写。隐式转换会导致索引无法使用，进而出现上述慢 SQL 堆积数据库连接数跑满的情况。

无索引案例

```
SELECT uid FROM `user` WHERE mo=13772556391 LIMIT 0,1
```

执行计划

```
mysql> explain  SELECT uid FROM `user` WHERE mo=13772556391 LIMIT 0,1;
           id: 1
```

```
  select_type: SIMPLE
        table: user
type: ALL
possible_keys: NULL
          key: NULL
         rows: 707250
        Extra: Using where
```

验证 mo 字段的过滤性

```
mysql> select count(*) from user where mo=13772556391;
|  0   |
```

添加索引

```
mysql> alter table user add index ind_mo(mo);
mysql>SELECT uid FROM `user` WHERE mo=13772556391 LIMIT 0,1;
Empty set (0.05 sec)
```

执行计划

```
mysql> explain  SELECT uid FROM `user` WHERE mo=13772556391 LIMIT 0,1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
         type: index
possible_keys: ind_mo
          key: ind_mo
         rows: 1
        Extra: Using where; Using index
```

隐式转换案例一

表结构

```
CREATE TABLE `user` (
  ......
mo char(11) NOT NULL DEFAULT '' ,
KEY ind_mo (mo)
......
) ENGINE=InnoDB;
```

执行计划

```
mysql> explain extended select uid from`user` where mo=13772556391 limit
0,1;
mysql> show warnings;
Warning1: Cannot use  index 'ind_mo' due to type or collation conversion
on field 'mo'
Note: select `user`.`uid` AS `uid` from `user` where (`user`.`mo` =
13772556391) limit 0,1
```

如何解决：

```
mysql> explain   SELECT uid FROM `user` WHERE mo='13772556391' LIMIT 0,1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
         type: ref
possible_keys: ind_mo
          key: ind_mo
      rows: 1
        Extra: Using where; Using index
```

隐式转换案例二

```
CREATE TABLE `test_date` (
    `id` int(11) DEFAULT NULL,
    `gmt_create` varchar(100) DEFAULT NULL,
    KEY `ind_gmt_create` (`gmt_create`)
) ENGINE=InnoDB AUTO_INCREMENT=524272;
```

5.5 版本执行计划

```
mysql> explain  select * from test_date where gmt_create BETWEEN
DATE_ADD(NOW(), INTERVAL - 1 MINUTE) AND  DATE_ADD(NOW(), INTERVAL 15
MINUTE) ;
+----+-------------+-----------+-------+---------------+------------
----+---------+------+------+-------------+
| id | select_type | table | type | possible_keys  | key | key_len | ref
| rows | Extra      |
+----+-------------+-----------+-------+---------------+------------
----+---------+------+------+-------------+
|1|SIMPLE| test_date |range| ind_gmt_create|ind_gmt_create|303| NULL |
1 | Using where |
```

5.6 版本执行计划

```
mysql> explain select * from test_date where gmt_create BETWEEN
DATE_ADD(NOW(), INTERVAL - 1 MINUTE) AND  DATE_ADD(NOW(), INTERVAL 15
MINUTE) ;
+----+-------------+-----------+------+---------------+------+------
---+------+---------+-------------+
| id | select_type | table | type | possible_keys  | key  | key_len | ref
| rows | Extra|
+----+-------------+-----------+------+---------------+------+------
---+------+---------+-------------+
```

```
   | 1 | SIMPLE| test_date | ALL | ind_gmt_create | NULL | NULL | NULL | 2849555
| Using where |
   +----+-------------+-----------+------+----------------+------+------
---+------+---------+-------------+
   |Warning|Cannot use range access on index 'ind_gmt_create' due to type
on field 'gmt_create'
```

隐式转换案例三

表结构

```
   CREATE TABLE `t1` (
     `c1` varchar(100) CHARACTER SET latin1 COLLATE latin1_bin DEFAULT NULL,
     `c2` varchar(100) DEFAULT NULL,
     KEY `ind_c1` (`c1`)
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
   CREATE TABLE `t2` (
     `c1` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin DEFAULT NULL,
     `c2` varchar(100) DEFAULT NULL,
     KEY `ind_c2` (`c2`)
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
```

执行计划

```
   mysql> explain    select t1.* from  t2 left  join  t1 on t1.c1=t2.c1 where
t2.c2='b';
   +----+-------------+-------+------+---------------+--------+---------
+-------+--------+-------------+
   | id | select_type | table | type | possible_keys |key| key_len | ref   |
rows   | Extra    |
   +----+-------------+-------+------+---------------+--------+---------
+-------+--------+-------------+
   | 1 | SIMPLE | t2 | ref  | ind_c2 | ind_c2 | 303     | const |   258 | Using
```

```
where |
   |1 |SIMPLE |t1  |ALL   | NULL   | NULL   | NULL    | NULL  | 402250 |    |
```

修改 COLLATE

```
   mysql> alter table t1 modify column c1 varchar(100) COLLATE utf8_bin ;
   Query OK, 401920 rows affected (2.79 sec)
   Records: 401920  Duplicates: 0  Warnings: 0
```

执行计划

```
   mysql> explain   select t1.* from  t2 left  join  t1 on t1.c1=t2.c1 where
t2.c2='b';
   +----+-------------+-------+------+---------------+--------+---------
+------------+-------+-------------+
   | id | select_type | table | type | possible_keys | key  | key_len | ref
| rows  | Extra       |
   +----+-------------+-------+------+---------------+--------+---------
+------------+-------+-------------+
   |  1 | SIMPLE| t2| ref | ind_c2| ind_c2 | 303     | const      |   258 | Using
where |
   |  1 |SIMPLE| t1|ref| ind_c1  | ind_c1 | 303     | test.t2.c1 | 33527 |
|
   +----+-------------+-------+------+---------------+--------+---------
+------------+-------+-------------+
```

两个索引的常见误区

误区一：对查询条件的每个字段建立单列索引，例如查询条件为：A=? and B=? and C=?

在表上创建了 3 个单列查询条件的索引 ind_A(A)，ind_B(B)，ind_C(C)，应该根据条件的过滤性，创建适当的单列索引或者组合索引。

误区二：对查询的所有字段建立组合索引，例如查询条件为 select A,B,C,D,E,F from T where G=?

在表上创建了 ind_A_B_C_D_E_F_G(A,B,C,D,E,F,G);

最佳实践

那如何避免无索引及发生了隐式转换呢？主要有以下途径：

在使用索引时，大家可以通过 explain+extended 查看 SQL 的执行计划，判断是否使用了索引以及发生了隐式转换。

由于常见的隐式转换是由字段数据类型以及 collation 定义不当导致，因此技术人员在设计开发阶段，要避免数据库字段定义，避免出现隐式转换。

由于 MySQL 不支持函数索引，在开发时要避免在查询条件加入函数，例如 date(gmt_create)。

所有上线的 SQL 都要经过严格的审核，创建合适的索引。

SQL 改写篇

SQL 优化在这里总结了三类常见的场景，包括分页优化、子查询优化、最佳实践。

分页优化：

表结构

```
CREATE TABLE `buyer` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
.....
```

```
  PRIMARY KEY (`id`)
KEY ind_seller (sellerid)
) ENGINE=InnoDB;
```

SQL 语句

```
Select * from buyer where sellerid=100 limit 100000, 5000
```

这条语句是普通的 Limit M、N 的翻页写法，在越往后翻页的过程中速度越慢，因为 MySQL 会读取表 M+N 条数据，M 越大，性能越差。

通过采用高效的 Limit 写法，可以将上述语句改写成：

```
Select t1.* from buyer t1,
(select id from buyer sellerid=100 limit 100000, 5000) t2
where t1.id=t2.id;
```

从而避免分页查询给数据库带来性能影响。需要注意一点是，这里需要在 t 表的 sellerid 字段上创建索引，id 为表的主键。

子查询优化：

典型子查询

```
SELECT first_name
 FROM employees
WHERE emp_no IN
(SELECT emp_no FROM salaries_2000 WHERE salary = 5000);
```

MySQL 的处理逻辑是遍历 employees 表中的每一条记录，代入到子查询中中去。

改写子查询：

```
SELECT first_name
FROM employees emp,
(SELECT emp_no FROM salaries_2000 WHERE salary = 5000) sal
WHERE emp.emp_no = sal.emp_no;
```

最佳实践：

采用高效的 Limit 写法，避免分页查询给数据库带来性能影响；

子查询在 5.1，5.5 版本中都存在较大风险，将子查询改为关联，使用 MySQL 5.6 的版本，可以避免麻烦的子查询改写；

另外避免用 Select * 查询所有字段数据，只查询需要的字段数据。

参数优化篇

数据库中的参数配置对 SQL 执行速度快慢也有非常大的影响，常见的三个参数包括 **innodb_buffer_pool_size**，**tmp_table_size**，**query_cache_size,table_cache**：

（1）innodb_buffer_pool_size

作用：定义了 innodb 引擎缓冲池的大小，该缓冲池主要缓存了索引以及数据，如果 SQL 查询的数据在缓冲池中已经缓存，那就不需要从磁盘中读取，性能会得到快速提升。笔者曾经看到有生产系统使用了 MySQL 的默认配置 128MB，导致数据库磁盘的使用率达到了 100%。

建议：通常配置主机内存的 70%~80%之间。

（2）tmp_table_size

作用：该参数用于决定内部内存临时表的最大值，每个线程都要分配

（实际起限制作用的是 tmp_table_size 和 max_heap_table_size 的最小值），如果内存临时表超出了限制，MySQL 就会自动地把它转化为基于磁盘的 MyISAM 表，优化查询语句的时候，要避免使用临时表，如果一定要使用临时表，要保证这些临时表是存在内存中的。

现象：如果复杂的 SQL 语句中包含了 Group by/Distinct 等不能通过索引进行优化而使用了临时表，则会导致 SQL 执行时间加长。

建议：如果应用中有很多 Group by/Distinct 等语句，同时数据库有足够的内存，可以增大 tmp_table_size(max_heap_table_size)的值，以此来提升查询性能。

（3）table_open_cache

作用：table_open_cache 指定表高速缓存的大小。每当 MySQL 访问一个表时，如果在表缓冲区中还有空间，该表就被打开并放入其中，这样可以更快地访问表内容。

现象：通常在设置 table_open_cache 参数的时候，在业务的高峰时期，检查 open_Tables 的值，如果 open_Tables 的值与 table_open_cache 的值相等，并且 opened_tales 的值在不断地增加，这个时候就需要对 table_open_cache 的值增加了，这个时候线程的状态为：Opening tables 。

```
mysql> show profiles;
| Query_ID | Duration   | Query          |
+----+----+------+
|      1 | 0.09211525 | select * from d |
|      2 | 0.03659925 | select * from d |
|      3 | 0.22665400 | select * from d |
|      4 | 0.11063350 | select * from d |
|      5 | 0.06929725 | select * from d |
|      6 | 0.09054975 | select * from d |
```

```
|        7 | 0.15971375 | select * from d |
|        8 | 0.12960625 | select * from d |
|        9 | 0.22713975 | select * from d |
|       10 | 0.00124025 | select * from d |
. mysql> show profile cpu for query 4;
+----------------------+----------+----------+------------+
| Status               | Duration | CPU_user | CPU_system |
+----------------------+----------+----------+------------+
| starting             | 0.000198 | 0.001000 |   0.000000 |
| checking permissions | 0.000053 | 0.000000 |   0.000000 |
| Opening tables       | 0.000454 | 0.000999 |   0.001000 |
| init                 | 0.000059 | 0.000000 |   0.000999 |
| System lock          | 0.000055 | 0.000000 |   0.000000 |
| optimizing           | 0.000053 | 0.000000 |   0.000000 |
| statistics           | 0.000056 | 0.000000 |   0.000000 |
| preparing            | 0.000056 | 0.000000 |   0.000000 |
| executing            | 0.000052 | 0.001000 |   0.000000 |
| Sending data         | 0.000072 | 0.000000 |   0.000000 |
| end                  | 0.000053 | 0.000000 |   0.000000 |
| query end            | 0.000056 | 0.000000 |   0.000000 |
| closing tables       | 0.000056 | 0.000000 |   0.000000 |
| freeing items        | 0.000076 | 0.000000 |   0.000000 |
| cleaning up          | 0.000056 | 0.000000 |   0.000000 |
```

通过这个 Profile 可以看到执行时间主要花费在 Opening tables，这个时候问题就比较清楚了，我们看一下 table_open_cache 这个参数的值是否较小，结果这个 RDS 的 table_open_cache 只有 100，而这个 RDS 却创建了上万张的表，进而导致了每次访问新的表的时候不得不重新打开，所以只需把 table_open_cache 调大即可解决目前的问题。

（4）query_cache_size

作用：该参数用于控制 MySQL Query Cache 的内存大小；如果 MySQL 开启 Query Cache，在执行每一个 Query 的时候会先锁住 Query Cache，然后判断是否存在 Query Cache 中，如果存在直接返回结果，如果不存在，则再进行引擎查询等操作；同时 Insert、Update 和 Delete 这样的操作都会将 Query Cache 失效掉，这种失效还包括结构或者索引的任何变化，缓存失效的维护代价较高，会给 MySQL 带来较大的压力，所以当数据库不是那么频繁的更新的时候，Query Cache 是个好东西，但是如果反过来，写入非常频繁，并集中在某几张表上的时候，那么 Query cachelock 的锁机制会造成很频繁的锁冲突，对于这一张表的写和读会互相等待 Query cache lock 解锁，导致 Select 的查询效率下降。

现象：数据库中有大量的连接状态为 Checking query cache for query、Waiting for query cache lock、Storing result in query cache。

建议：RDS 默认是关闭 Query Cache 功能的，如果您的实例打开了 Query Cache，当出现上述情况后可以关闭 Query Cache；当然有些情况也可以打开 Query Cache，比如：巧用 Query Cache 解决数据库性能问题。

优化器篇

优化器根据统计信息以及优化器参数计算出 SQL 的执行计划，所以统计信息和优化算法决定着执行计划的优劣。常见优化器导致 SQL 执行出现缓慢的情况包括两种，统计信息不准确导致索引走错出现性能下降；数据库版本升级导致优化器参数发生变化，进而导致执行计划发生变化，性能可能变差。

优化器参数

数据库从 5.5 升级到 5.6，一条 SQL 在 5.5 执行只需要零点几秒，而在 5.6 上需要 10 多秒。

（1）5.5 的优化器策略：

```
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_m
erge_intersection=on,engine_condition_pushdown=on
```

（2）5.6 的优化器策略：

```
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_m
erge_intersection=on,engine_condition_pushdown=on,block_nested_loop=on…..
mysql> explain SELECT *
    -> FROM t1 this_
    -> LEFT OUTER JOIN t2 item2_ ON this_.itemId = gameitem2_.id
    -> LEFT OUTER JOIN t3 group3_ ON gameitem2_.groupId =gamegroup3_.id
       …….
    -> LEFT OUTER JOIN t8 leagueitem10_ ON leagueinfo7_.itemId
=leagueitem10_.id
    -> ORDER BY this_.id ASC LIMIT 20;
+----+-------------+---------------+--------+---------------+--------
-+---------+-----------------------------+--------+----------------------
------------------+
| id | select_type| table| type|possible_keys | key| key_len | ref   | rows
| Extra  |
+----+----------+---------------+--------+---------------+---------+-
--------+--------------+-----------------+
| 1 | SIMPLE| this_| ALL|NULL|NULL|NULL|NULL|257312|Using temporary;
Using filesort |
| 1 |SIMPLE|item2_|eq_ref|PRIMARY|PRIMARY|4|this_.itemId|1|NULL
| 1 | SIMPLE| group3_| ALL | PRIMARY| NULL| NULL| NULL | 6 |Using where;
```

```
Using join buffer (Block Nested Loop)

   mysql> set optimizer_switch='....block_nested_loop=off....';

   +----+-------------+--------------+--------+---------------+--------
-+---------+-------------------------------+------+--------+
   | id | select_type | table |type | possible_keys | key     | key_len | ref
| rows | Extra |
   +----+--------------+--------------+--------+---------------+------
---+---------+-------------+------+--------+
   | 1 | SIMPL | this| index   | NULL  | PRIMARY | 4   | NULL |   20 | NULL
|
   | 1|SIMPLE| item2_| eq_ref | PRIMARY| PRIMARY | 4 | this_.itemId |  1 |
NULL  |
   | 1|SIMPLE | group3_ | eq_ref | PRIMARY | PRIMARY | 4 | item2_.groupId
|   1 | NULL |
```

统计信息

```
   CREATE TABLE `t1` (
     `c1` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin DEFAULT NULL,
     `c2` varchar(100) DEFAULT NULL,
     KEY `ind_c1` (`c1`)
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8

   mysql> explain  select * from t1  where c1='m';
   +----+-------------+-------+------+---------------+------+---------+-
-----+--------+-------------+
   | id | select_type | table | type | possible_keys | key  | key_len | ref
| rows | Extra      |
   +----+-------------+-------+------+---------------+------+---------+-
-----+--------+-------------+
   | 1 | SIMPLE| t1    | ALL  | NULL | NULL | NULL    | NULL | 804273 | Using
```

```
where |


   mysql> show index from t1;
   | Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation
| Cardinality | Sub_part | Packed | Null | Index_type | Comment |
   +-------+------------+----------+--------------+-------------+-------
----+-------------+----------+--------+------+------
   | t1 |  1 | ind_c1   |   1 | c1  | A  |  0 |   NULL | NULL | YES  | BTREE   |
|
   +-------+------------+----------+--------------+-------------+-------
----+-------------+----------+--------+------+-------

   mysql> analyze table t1;
   +---------+---------+----------+----------+
   | Table   | Op      | Msg_type | Msg_text |
   +---------+---------+----------+----------+
   | test.t1 | analyze | status   | OK       |
   +---------+---------+----------+----------+

   mysql> show index from t1;
   | Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation
| Cardinality | Sub_part | Packed | Null | Index_type | Comment |
   +-------+------------+----------+--------------+-------------+-------
----+-------------+----------+--------+------+-------
   | t1 | 1 | ind_c1|  1 | c1 | A   |  18 |   NULL | NULL   | YES  | BTREE     |
|

   mysql> explain  select * from t1 where c1='m';
   +----+-------------+-------+------+---------------+--------+---------
+-------+------+-------------+
   | id | select_type | table | type | possible_keys | key  | key_len | ref
```

```
| rows | Extra      |
   +----+-------------+-------+------+---------------+--------+---------
+-------+------+-------------+
   |  1 | SIMPLE| t1    | ref  | ind_c1| ind_c1 | 303     | const |    1 | Using
where |
   +----+-------------+-------+------+---------------+--------+---------
+-------+------+-------------+
```

小结

SQL 的优化是 DBA 开发日常工作中不可缺少的一部分，曾经有人用一个公式来讲解他在做 SQL 优化的时候遵循的原则：

T=S/V(T 代表时间，S 代表路程，V 代表速度)

S 指 SQL 所需访问的资源总量，V 指 SQL 单位时间所能访问的资源量，T 自然就是 SQL 执行所需时间了;为了获得 SQL 最快的执行时间，可以根据公式定义上去反推：

（1）在 S 不变的情况下，可以提升 V 来降低 T：通过适当的索引调整，可以将大量的速度较慢的随机 IO 转换为速度较快的顺序 IO；通过提升服务器的内存，使得将更多的数据放到内存中，会比数据放到磁盘上会得到明显的速度提升；采用电子存储介质进行数据存储和读取的 SSD，突破了传统机械硬盘的性能瓶颈，使其拥有极高的存储性能;在提升 V 上可以采用较高配置的硬件来完成速度的提升；

（2）在 V 不变的情况下，可以减小 S 来降低 T：这是 SQL 优化中非常核心的一个环节，在减小 S 环节上，DBA 可以做的可以有很多，通常可以在查询条件中建立适当的索引，来避免全表扫描；也可以通过改写 SQl，添加一些适当的提示符，来改变 SQL 的执行计划，使 SQL 以最少的扫描路

径完成查询；当这些方法都使用完了之后，你是否还有其他方案来优化？

在阿里系的 DBA 职位描述中有条要求是 DBA 需要深入了解业务，当 DBA 深入的了解业务之后，既能站在业务上，又能站在 DB 角度考虑，这个时候再去做优化，有时候能达到事半功倍的效果。

第4章 业务研发经典案例

在阿里巴巴集团，有大量的业务线技术研发人员，在业务线的研发人员每天都会遇到各类技术问题。业务线的问题往往都隐藏在业务和技术结合的系统抽象和系统流程中，比如锁的问题，事务问题，缓存问题。业务线的技术问题一般排查的链路比较长，从浏览器到网络，到 web 服务器，到服务化应用，到缓存，再到存储。作为一名合格的研发人员，必须掌握一些线上问题排查技巧。接下来本章节会从业务技术视角来介绍几个经典的案例，从问题发现，问题 root cause 分析，解决方案实施，效果检查几个方面着手，希望其中的排查思路和解决方案能对读者带来启发。

4.1 幂等控制，分布式锁超时情况和业务重试的并发

背景说明

某日，做产品 X 的开发接到客户公司电话，说是对账出了 1 分钱的差错，无法处理。本着“客户第一”的宗旨，开发立马上线查看情况。查完发现，按照产品 X 当日的年化收益率，正常情况下用户在转入 57 元后一共收益 3 分钱，合计是 57.03 元。但是该客户当日却有一笔消费 57.04 元，导致客户公司系统对多出的 1 分钱处理不了。再进一步分析，发现用户收益结转时多了 1 分钱的收益，并且已消费……

也就是说，本来用户只有 3 分钱收益，结果多发了 1 分钱给他，也就给公司造成 1 分钱的损失！用户在产品 X 里当天收益本应该是 0.03 元，怎么会变成 0.04 元呢？多出的 1 分钱收益从哪里来的呢？

数据库记录分析

带着上面的一系列疑问，开发人员首先排查了产品 X 收益的数据库记录。通过查询数据库发现，该用户收益结转在同一天内存在 2 笔交易记录。交易记录 1 创建时间为 8:00:23，记录 2 创建时间为 8:00:29，交易记录 1 和 2 的最后修改时间均为 8:00:29，如图 4-1 所示。

USER_ID	TXID	收益	创建时间	修改时间
用户A	TXID a	0.03	8:00:23	8:00:29
用户A	TXID b	0.03	8:00:29	8:00:29

TXID不同，上层业务重试

创建时间早6秒，该笔存在超时情况

修改时间相同，同时提交数据库

图 4-1　用户当日收益结转数据库记录分析

正常情况下产品 X 收益每天只会结转一次，而这个用户当日有两笔收益结转记录。开发人员怀疑，很可能是出现了并发问题。

继续跟踪第一笔“TXID a”的记录，开发确认线上日志存在超时情况，失败原因是数据库链接数已满，线程等待提交。

分布式锁超时时间是 5s，第一笔记录从创建到修改提交经历了 6s，由此可见是在分布式锁失效之后，获得了数据库链接，进行提交成功。

有了以上三个排查思路后，我们可以开始逆推整个过程。

过程逆推

根据数据库记录逆推当时的运行情况，如图 4-2 所示。

（1）由于数据库连接数被占满，流水 1 创建的事务处于等待提交状态。

（2）系统 A 发现交易失败，重试次数不满 8 次的，立即发起重试，触发生成流水 2 的请求。

（3）5s 以内数据均被分布式锁拦截，无法提交。

（4）经过 5s 后，系统 B 的分布式锁失效，此时事务仍在等待未提交。

（5）6s 时，流水 2 成功越过数据库查询幂等校验发起事务，此时流水 1 拿到数据库连接，流水 1 和 2 两个事务同时提交。

（6）由于数据库未做唯一索引，且支付受理模块打穿下层幂等原则，生成 2 个 TXID，导致两事务同时提交成功。

（7）收益结转重复记账，用户多了一笔收入。

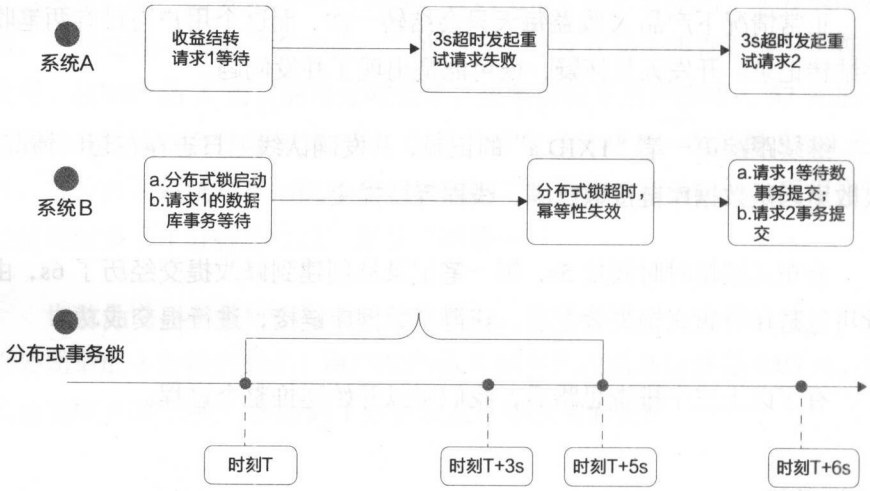


图 4-2　数据库分布式锁超时并发控制失效

深入分析

完成了整个问题的过程逆推后，开发人员进一步分析，发现问题真正的原因还是在系统设计上。如图 4-3 所示，系统 A 的事务允许一定时间的等待，而上层业务的重试时间又比这个等待的时间要短。这就存在一个问题：系统 A 的事务还在等待中，业务就又发起了重试。如果是在这个应用场景下（可能业务上对重试要求更高一些），那么对幂等控制的要求就更高了。而仅仅通过一个分布式锁来控制，如果分布式锁的超时时间设置的比事务允许等待的时间短，那么在锁失效之后就一定会同时提交两笔请求。

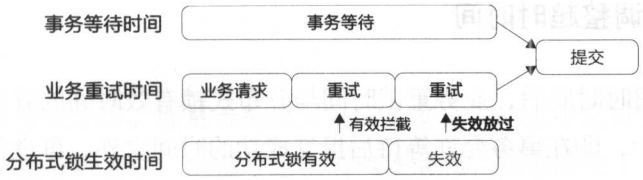


图 4-3　分布式锁超时并发控制时间轴

继续对整个过程抽象化，开发人员得出一个结论：分布式锁在以下条件同时满足的情况下并发控制会被打穿。

（1）上层业务系统层面有重试机制。

（2）业务请求存在一定时间之后提交成功的情况，例如本例中第一次请求在事务等待 6s 后获得了数据库链接，提交数据库成功。

（3）下游系统缺乏其他有效的幂等控制手段。

思考

了解了问题的来龙去脉后，接下来要怎么解决这类问题呢？我们想了以下几个方案。

（1）调整 B 系统上的 tr 和分布式锁超时时间，tr 超时调整为 10s，分布式锁超时调整为 30s。

（2）防止做收益结转产生并发控制幂等，调整了收益结转流水号的生成规则：前 8 位取 X 收益结转传入的交易号的前 8 位，第 10 位系统版本设置为“9”，最后 8 位 seq 取交易号的最后 8 位，降低问题出现几率。

方案一：调整超时时间

调整超时时间后，业务重试时间与分布式锁有效时间的分布时间轴如图 4-4 所示，即在事务允许等待后提交成功的时间之外，再进行重试，另外分布式锁在整个阶段均有效，防止提交。

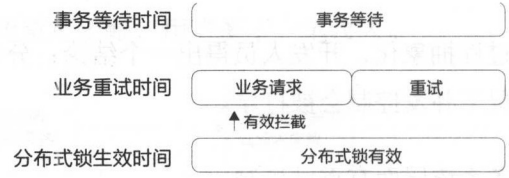


图 4-4 分布式锁超时并发控制时间轴

方案一验证有效。

方案二：增加幂等控制（推荐）

如图 4-5 所示，单纯靠分布式锁不是控制并发幂等的方式，最稳妥的方式还是在提交记录的时候通过数据库严格控制幂等。确保不论如何设置超时时间，都不会出现幂等控制的问题。

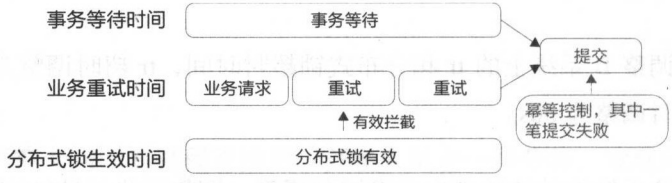


图 4-5 分布式锁超时并发控制时间轴

方案二验证有效。

小结

资金安全无小事，而幂等控制又是资金安全中的重中之重。回顾本文案例，从问题分析定位，到整个逻辑的梳理清洗，其中涉及了三个时间轴的相互作用，再加上事务、分布式锁、重试等，整个问题发生的逻辑还是比较复杂的。因此，在系统并发幂等控制设计中，单纯的分布式锁并不具备严格控制并发幂等的作用，建议在系统设计时，将第三方唯一性的幂等控制作为幂等控制的兜底方案，控制好这道幂等防线，这样不论业务如何设计，就万变不离其宗了。

4.2　另类解法，分布式一致性

背景

在大型互联网系统中，基于成本的考虑，普遍会使用 MySQL 数据库；同时由于业务量很大，通常会按照用户维度对数据做垂直拆分，即大家常说的分库分表。

在阿里巴巴的红包系统中，红包的发放操作会涉及两个数据库的事务操作，一个数据库进行预算的扣减，另一个进行用户红包数据的写入，那么如何保证这两个事务操作的一致性呢？

问题原因及分析

开发人员首先想到的就是使用 MySQL 的 XA 协议，它使用的是两阶段提交协议，如图 4-6 所示。由事务协调者来保证所有的事务参与者都完成了第一阶段的准备工作，如果所有参与者都准备好了，那么就通知所有

的参与者进行提交。MySQL 数据库仅能扮演参与者的角色，协调者（事务管理器）需要由另外的应用担任。

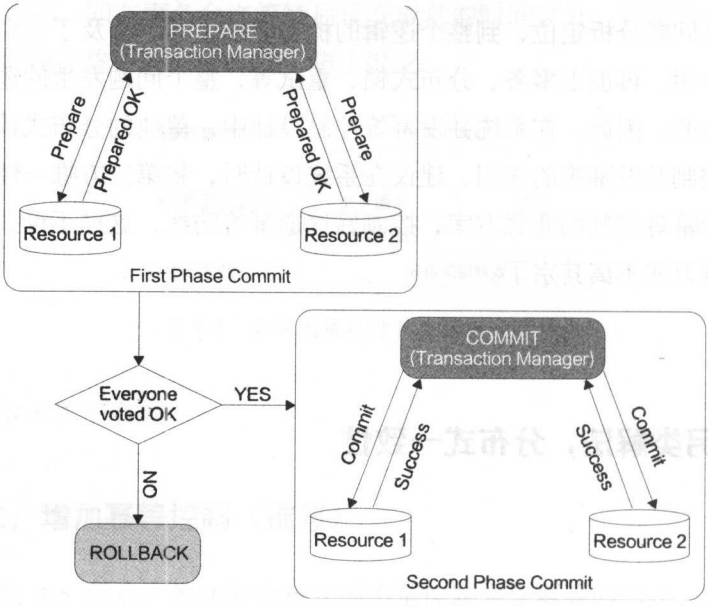


图 4-6 两阶段提交协议

两阶段提交协议对于红包系统来说，在 XA 事务中的第一阶段，预算端需要做的事情是冻结要扣除的预算，红包数据端需要插入一条不可用的红包数据。在 XA 事务的第二阶段，如果事务需要提交，那么预算端需要将第一阶段的冻结的预算转化为实际的扣除，红包数据端需要将刚才插入的那一条不可用的红包数据变为可用；如果事务需要回滚，那么预处端需要将第一阶段的冻结的预算进行撤销操作，红包数据端需要删除刚才插入的不可用的红包数据。

对于任何一次不论是成功还是失败的发放操作，预算端都会涉及两次针对于预算纪录的事务操作。众所周知的是，该条记录的操作会存在热点写的问题，使用 XA 事务的方案会将热点写问题放大一倍。在业务场景中，

使用 XA 事务的方案看起来有些过于“悲观”了，事实上红包发放的成功与否几乎完全取决于预算是否扣减成功。回到业务的根本需求上来，有以下两个关键点。

（1）希望预算扣减成功与产生用户红包数据这两件事要么同时成功，要么同时失败，只要预算足够，那么理论上是肯定成功的。

（2）不可能将每一个发放操作变成一个一个顺序的独立事务，这样不但性能非常差，同时后面的事务也并不关心预算的具体剩余金额。所以实际上在发放操作中，很有可能看到一个是中间态的预算剩余金额，这是允许的。

如果能存在一起机制，它能保证预算扣减成功后 100%会通知到我们，这样就可以利用这个机会去产生用户红包数据，如果产生失败，那么我们可以对已经扣减的预算进行一笔反向操作。

解决过程

最终的解决方案是开发人员设计了一个轻型的一致性消息组件，把预算扣减成功等诸多业务操作事件写入数据库中，该消息组件保证事件与业务操作处在同一个数据库中，所以仅仅是一次简单的本地事务操作。该消息组件会对写入的事件进行分发，通知每一个事件订阅者（比如在当前的场景中就是进行用户红包数据写入），然后对事件的处理结果进行记录。

在一个成功的红包发放操作中，对于预算端来说，进行了一次预算的扣减，一次事件写入，可能会进行一次事件读取，一次事件状态更新；对于用户红包数据端，仅需要进行一次用户红包数据的写入。可以看到，对于预算扣减的热点数据操作，新的方案并没有放大它。同时，失败的红包发放操作就更简单了，仅仅是一次失败的预算扣减，没有事件，没有用户红包端的任何操作。

另外，在实际使用中开发人员发现，使用这样的一套方案还可以降低系统的编码以及运维的复杂度，不再需要为预算端设计冻结操作以及用户红包数据端设计不可用数据，也不需要引入另外一套独立的事务协调系统。

这套系统命名为 MiniBus，即微型总线，如图 4-7 所示。它包括以下三部分。

（1）事件发布者 Publisher。由业务逻辑在业务事务中调用，将事件写入到与业务相同的数据库中。

（2）事件配置 Metadata。管理事件的基本信息，订阅关系等。

（3）事件调试器 Scheduler。读取待分发的事件，根据订阅关系进行分发，调用相应的事件处理器，然后记录事件分发的结果。

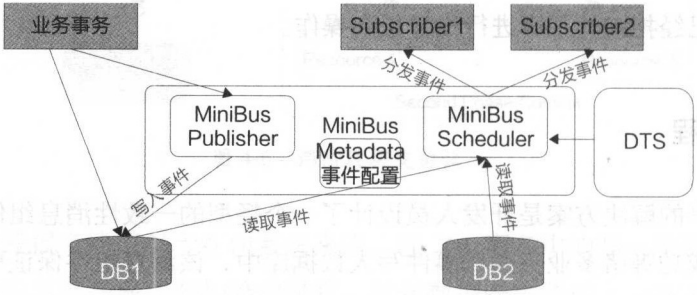


图 4-7 微型消息总线系统架构

该方案并不仅仅解决了红包发放操作中的事务一致性问题，事实上，只要业务上满足如下的两个要求，它都适用。

（1）短暂的中间状态是可以接受的。

（2）要求最终一终性。

同时，还有如下 4 个方面的优点。

（1）较 XA 方案更易于编码和理解，已有代码的改造成本很低。

（2）数据库消耗增加较少。

（3）系统简单，外部依赖少，易于维护。

（4）兼容数据库上的其他他 SQL 优化。

小结

好的技术方案都是对于业务场景的深刻理解和权衡之后获得的，虽然通用的方案是首选，但它并不是唯一的好的选择。

4.3　大道至简，从故障模型的边界状态切换到原始状态

背景

庄子云：“吾生也有涯，而知也无涯。以有涯随无涯，殆已！”诚如斯言，分布式场景中完善的容错机制设计，需要考虑好网络异常、磁盘 IO 异常、时钟跳变、操作系统异常，乃至软件本身可能存在 bug 缺陷等诸多边缘情况。面临上述情况，如何确保分布式系统能够正确运行是极大的挑战。

我们总是会面临自己的分布式系统在某些局部故障下运行状态是很难说被考虑完善无误的，即使增加更多严格测试用例，依旧存在遗漏的可能性，进而导致我们的系统进入非预期状态的严重后果。在这里，一个切实可操作的、可以被用来简化边缘情况下的复杂场景，消除未来潜在不可预期状态的容错机制设计指导原则是“Convert Partial Failure to Fail-Stop Failure”，即将分布式系统中发生的无法处理的局部故障直接转换为标准流

程中的失败，也即停止故障。

某日上午 11 点左右，飞天女娲值班人员接到报警，显示 ECS 某个集群内全部女娲 Proxy Servers 服务地址解析请求超时，很快存在有设备挂载到该集群的该区域内其他集群陆续报警发生虚拟机 IO Hang。ECS 运维人员以及女娲研发人员排查确认该集群内女娲一致性服务组的主服务器处于机器 Hang 状态，无法直接登录，运维人员重启这台女娲主服务器，女娲地址解析功能随即恢复正常，该区域内发生 IO Hang 的虚拟机也逐步全部恢复。

作为飞天系统核心基础模块之一，女娲对阿里云众多云产品提供了分布锁、地址注册解析、元数据存储等分布式协同服务。就地址解析服务而言，女娲的服务架构如图 4-8 所示，后端的一致性服务组（Consensus Core）基于 Paxos 协议实现，由奇数个服务器（我们称之为 Quorum Server，不妨假定数目为 n）组成的高可用的一致性核心。通过选举，一致性服务组会产生 1 个主服务器，以及（n-1）个从服务器。无论是主服务器还是从服务器，其均可以响应客户端的读写等非事务操作请求，但是对于写等改变一致性系统状态的事务操作请求，从服务器需要转发给主服务器处理，由主服务器在一致性服务组内部发起接受该事务操作请求的提议。中间代理层（Proxy Layer）则是面向客户端提供了直接的高扩展地址解析请求的代理服务，其由多个无状态代理服务器组成，每个代理服务器均基于订阅通知实现了独立、高效的缓存。女娲的 SDK 则提供了地址解析接口，用户通过调用该接口，可以解析其业务 Master 注册过的女娲服务地址，以获取其业务 Master 当前工作的 TCP 地址。

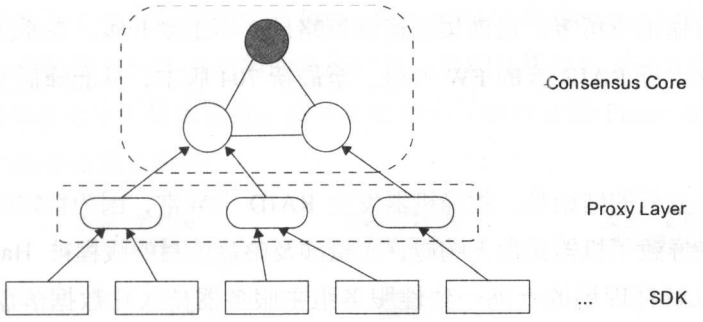


图 4-8　女娲地址解析服务服务架构

理论上，基于 Paxos 实现的女娲一致性服务组，是可以容忍单个服务器的机器故障，继续提供分布式协同服务的，但是事实上该集群的女娲一致性服务组的主服务器发生机器 Hang 之后，集群女娲即停止了服务，从而导致了业务大量虚拟机发生 IO Hang。Hang 这个是本次故障调查初期，参与故障处理的女娲研发人员心里最直接的困惑

调查

问题 1：该集群的女娲一致性服务组主服务器发生机器 Hang 的原因？

阿里云硬件团队介入调查，并很快确认发生故障的机器为 HP 服务器 6CU313YDQ0，机型为 D10，导致机器 Hang 的原因是 RAID 卡故障，在 HP IML 日志中找到相关错误日志信息：Drive Array Controller Failure (Slot 0)。事实上，有关 HP RAID 卡的该类型故障，最早出现在集团 DB，集团 DB 使用 6U 的操作系统，驱动为 HPSA，在出现同类型故障的时候会在驱动层面直接宕机。而在 5U 操作系统里面驱动使用的是 CCISS，而非 HPSA，在出现同类型 RAID 卡故障时候表现为：设备不宕机，IO 异常，重启硬件后故障现象消失。

这个问题从 2016 年 3 月份开始发生，期间硬件团队也有接 RAID 卡的

串口进行输出分析等。目前集团运营策略是，不主动升级，在系统重装情况下自动进行 RAID 卡的 FW 升级，至最新 704 版本，从而降低发生问题的概率。

上述分析的结论是，故障机器发生 RAID 卡异常，因为特定机型以及相关驱动导致了机器磁盘无响应，大量涉及磁盘读写的线程被 Hang 住。这个可以解释现场的女娲一致性服务组主服务器持久化数据落盘线程被 Hang 住，该服务器上的女娲代理服务进程，一致性服务进程的日志输出均缺失，操作系统日志、机器性能监控日志、内核日志等均缺失的现象。

值得关注的是，该集群的女娲一致性服务组主服务器 RAID 卡异常发生的时间在更早之前，持续了将近两天时间（期间女娲地址解析服务正常），最终才触发故障。这将近两天时间里面，因为问题机器磁盘无响应，机器上监控指标采集线程同样被 Hang 住，无法汇报女娲针对女娲一致性服务组中主服务器的服务状态监控采集的信息，这个也导致开发人员没有能够及时收到报警，从而无法在第一时间上线处理。

这是技术人员在这个故障里面得到的第一个惨痛教训：即使是监控脚本，也需要具备冗余容错能力。故障发生之后，很快在所有集群 Admin Gateway 机器上，以及女娲 Quorum Server 机器上部署了功能类似的监控脚本，涉及女娲一致性服务，日志输出，磁盘健康状况乃至女娲一致性服务组中所有服务器的机器网络状况等多项监控，确保集群网关登录机器与女娲一致性服务组中服务器在女娲服务监控这块有冗余容错能力。

问题 2：该集群女娲一致性服务组状态是否正常以及对地址解析服务影响是？

该集群内的女娲 Proxy Servers 与 Quorum Servers 为混合部署。为了描述方便，如图 4-9 所示，我们不妨将发生故障的集群里面三台女娲 Quorum Servers 机器称为 Q1、Q2、Q3，将三台 Proxy Servers 称为 P1、P2、P3，

其中故障机器上混合部署的女娲 Proxy Server P2 以及 Quorum Server Q2 的日志，因为机器 RAID 卡故障而全部缺失，我们从集群里面另外两台女娲 Proxy 的日志上分析具体影响。图 4-9 即展示了当时女娲 Proxy Server P1 地址解析服务受到的影响。

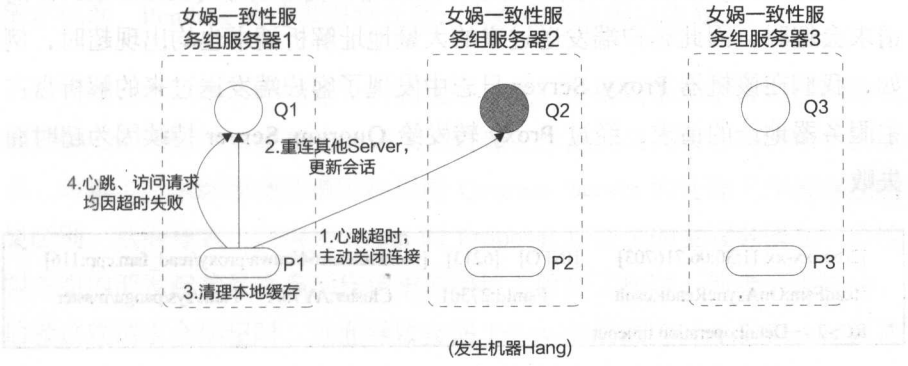


图 4-9　女娲地址解析服务的被影响过程

首先，女娲 Proxy Server P1 因与后端 Quorum Server Q2 超过了指定时间阈值没有心跳包回复，因此认定两者之间的会话过去，主动关闭了连接，并尝试重新连接另一台 Quorum Server Q1，最终成功建立连接，并且更新了会话。注意，女娲 Proxy Servers 提供基于订阅通知实现的高效缓存，在与后端重连期间，本地缓存依旧有效，即地址解析服务正常。而当 Proxy Server P1 与后端 Quorum Server Q1 成功建立连接，并且更新会话之后，将本地缓存所有缓存项清理干净（断开连接期间，后端 Quorum Server 的女娲地址内存可能发生改变，清理 Proxy 本地缓存是为了使得其后续可以根据客户端的访问请求重新更新本地缓存为当前最新内容），然后通知所有关心的客户端来重新获取相关女娲地址最新内容。

事实上，此时 Proxy Server P1 与后端 Quorum Server Q1 并未“真正”建立正确、合法的连接。后续的心跳请求还是会超时，进而导致 Proxy Server 继续关闭了连接，重连 Quorum Server Q3，后面就陷入了“与当前 Quorum

Server 心跳超时→关闭与当前 Quorum Server 的连接→建立与其他 Quorum Server 连接→成功更新会话→清理本地缓存→与 Quorum Server 心跳、请求超时→……”这样的循环中。

因为 Proxy Server P1 的本地缓存被清理，发给后端 Quorum Server 的请求会超时，因此客户端发送过来的大量地址解析请求也均出现超时，例如，我们在该机器 Proxy Server 日志中发现了客户端发送过来的解析盘古主服务器地址的请求，经过 Proxy 转发给 Quorum Server 持续因为超时而失败。

```
[20xx-xx-xx 11:30:06.716703]  [INFO]  [6213]  [build/release64/nuwa/proxy/read_fsm.cpp:116]
ReadFsm:OnAsyncReadResult  FsmId:27301  Cluster:AY76A  Path:/sys/pangu/master
RC:-7  Detail:operation timeout
```

在另一台女娲 Proxy Server P3 的日志中，技术人员观察到了完全类似的现象。也就是说，可以确定导致本次故障最直接的原因是提供地址解析服务的女娲 Proxy Servers 本地缓存被清空，其访问后端 Quorum Servers 的请求因为超时而失败。

导致女娲 Proxy Server 与 Quorum Server 之间的心跳、读等访问请求持续超时的原因，通过分析剩余两台 Quorum Servers（Q1 与 Q3）的审计日志可以进一步定位。以 Q3 上的日志为例，其 11:09 收到了一个新建会话的请求，然后一直等到运维人员重启了出问题的 Quorum Server 机器，恢复了女娲服务之后，审计日志才出现了该请求被处理完成的信息，即如果没有人为主动重启机器，恢复女娲服务，看起来这个请求被阻塞时间仍要继续持续下去。

也就是说，这个阶段发送给 Quorum Server 的所有心跳请求、读请求以及新建会话等请求均已被阻塞，在后端得不到处理，这个导致了集群内女娲 Proxy Servers 与后端女娲 Quorum Servers 的连接处于不正常状态。如

图 4-10 所示，通过观察故障期间的该 ECS 集群的女娲后端 Quorum Servers 的神农性能监控，我们发现大量访问请求在女娲后端 Quorum 得不到处理，峰值 Pending 的请求数达到了 6.5KB，即经过女娲 Proxy Servers 发送的读访问请求均 Pending 在后端得不到处理。最终重启 Quorum Leader Server 所在机器，Pending 的请求得到处理，女娲服务也得以恢复。

表面上看，第二个问题分析到现在，我们会得到第二个教训点，即女娲后端 Quorum 针对更新会话请求的处理工作流，与其余心跳请求、读请求、写请求乃至新建会话请求在后端 Quorum Server 的处理工作流存在明显区别，从而导致当后端 Quorum 的 Proposal 功能（即主服务器在一致性服务组内部发起接受事务操作请求的提议）暂停工作时，前者请求成功，后者这些请求全部超时，进而导致女娲 Proxy 与后端 Quorum Server 重建了“非健康状态”的连接，清理了原有旧的缓存，触发了本次故障。

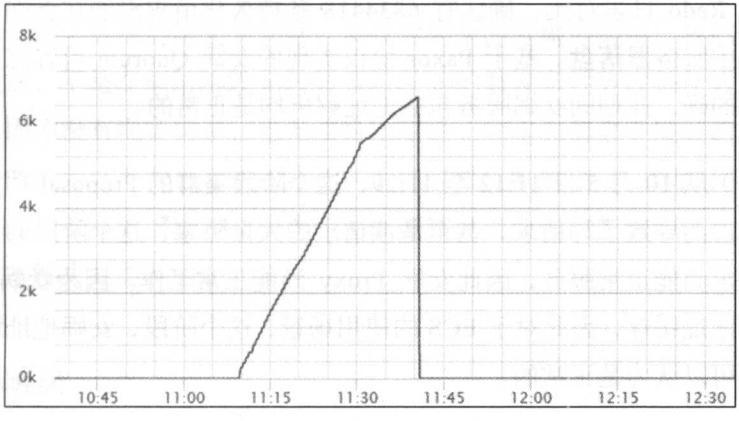


图 4-10　故障发生前后，女娲 Quorum Servers 请求堆积情况

但是，就问题根源来说，我们在这个故障里面接受到的第二个惨痛教训事实上是：女娲 Proxy Server 提供了集群内地址解析服务，其主动清理掉状态不确定的缓存项并不合适。从服务健壮性角度考虑，更加合理的做法是应该在每个缓存项至后端 Quorum Server 的访问请求成功返回后，主

动更新本地缓存项。基于这样的思考，在故障发生之后，我们从女娲 Proxy Server 的缓存，乃至 SDK 内部缓存等多级缓存中，均在订阅通知确保正确性的前提下，当 Proxy Servers 或者 Quorum Servers 工作不正常情况下，能够对业务提供本地缓存的访问，提升服务健壮性。

问题 3：女娲一致性服务组主服务器服务 Hang 住为什么会导致整个一致性服务组的请求处理被阻塞？

根据后续对该故障的全面复盘，事实上，整个故障发生时持续了以下三个阶段。

（1）从 10 月 3 日 14:38 至 10 月 5 日 11:11，这个阶段集群的女娲一致性服务组主服务器发生了 RAID 故障导致的磁盘 IO Hang。这个导致了发送到 Quorum 的写等事务性请求在该机器上均无法落盘，根据故障前后持久化的 Redo 日志对比，确认有 6834418 条持久化请求没能在女娲一致性服务组主服务器落盘。基于 Paxos 协议实现的女娲 Quorum 可容忍单节点故障，的确，此期间女娲服务包括地址解析均是正常的。

（2）从 10 月 5 日 11:12 至 11:29，这个阶段集群的 Proposal 功能开始阻塞，进而导致了写请求，甚至是读请求的大量阻塞，这个阶段的女娲的心跳请求尚能正常服务，因此女娲 Proxy 尚能正常工作。因为女娲 Proxy 管理这地址缓存，因此对于 ECS 的使用场景，这个阶段，女娲地址解析服务依然可以认为是正常的。

（3）从 11:30 开始，这个阶段，女娲后端 Quorum 处理心跳也会阻塞，从而导致了女娲 Proxy 意识到连接状态异常，尝试重新连接，并清理本地缓存，导致了业务重新发起的解析盘古主服务器地址超时失败，导致故障。

如上面分析的，阶段（1）是预期中的，不再赘述。有关阶段（2）、（3）的分析，事实上，整个 Quorum 的 Proposal 功能被阻塞的，那么相应的读

等非事务性的请求，以及心跳请求都会被陆续阻塞。这个是可以解释的，在女娲 Quroum Server 的代码实现中，为了维持单节点上看到的读、写请求的顺序性而实现的，即上一个经过本 Quorum Server 发起的事务性请求没有返回，那么后续到这个 Quorum Server 的非事务性请求是会阻塞住的。

所以，剩余一个不明确的点在于：为什么女娲一致性服务组主服务器机器 Hang，会最终导致整个 Quorum 的 Proposal 功能被阻塞，即阶段（1）至阶段（2）的转换条件是什么。因为现场日志的缺失，这个问题只能做个推测：女娲一致性服务组主服务器在阶段（1）近两天时间里面，累积的海量的的事务性请求，这些请求均阻塞着等待落盘，这些过多小的请求对象累积、堆积最终导致了 JVM 内存回收时间变长，进而请求处理变得十分缓慢，从现象上观察就是女娲一致性服务组主服务器的 Proposal 功能被阻塞住。

关于这个关键的第三个问题，调查后的全面复盘讨论中，我们后面关注的重点，不是在纠结阶段 a)转换至阶段 b)的触发条件到底是什么，这个事实上因为没有现场信息，也无法给出定论。如本文前言中提到的，我们更关注的还是这个背后反映的分布式系统容错机制的设计，即如何将触发本故障的局部故障场景转换成我们可以能够完美处理的失败即停止场景，从而根本解决此类问题。

讨论&解决

在本次故障中，最终 ECS 运维人员重启集群女娲一致性服务组主服务器，这个就是一个人为的失败即停止行为，正是这个操作很快恢复了该集群的飞天女娲服务以及 ECS 业务服务。这里我们围绕着“Convert Partial Failure to Fail-Stop Failure”意义，适当展开下讨论。

分布式系统中的错误类型复杂多样，业界流行的一致性协议（如 Paxos）

都是基于 crash failure（i.e., failure-stop）的假设，也就是说 crash failure 不会影响协议的正确执行。所以我们在系统设计上将 partial failure 转换为 crash failure 后就会将故障转换为协议可以处理的场景。在某些特定场景下，我们其实是可以容忍 partial failure，让系统进入 graceful downgrade。我们目前也正在特定功能的 graceful downgrade 策略上进行研发。

那么，如何界定故障中的“Partial Failure”？如何给出我们可以准确 Handle 的“Failure-Stop Failure”？这两者之间的转换、触发条件又是什么？下面是我们具体解决思路。

（1）判断故障中的女娲一致性服务组主服务器的“Partial Failure”

就本次故障而言，体现出来的女娲一致性服务组主服务器需要处理“Partial Failure”是事务性请求关联的 Redo 数据落盘操作 Hang 的场景。女娲 Quorum Server 对于请求是管道形式来处理的，就等待落盘的事务性请求来说，Quorum Server 内部有维持一个队列。

因此，在确认存放待落盘事务性请求的队列中存在请求项的前提下，通过记录一致性服务组主服务器上次事务性请求落盘的时间，对比当前时间，可以判断出本一致性服务组主服务器数据落盘 Hang 住的“Partial Failure”。

（2）根解问题的女娲一致性服务组的“Fail-Stop Failure”

当女娲一致性服务组主服务器退出自身 LEADING 状态（通过主动退出角色，或者直接退出进程方式），可以让剩余的女娲一致性服务组从服务器意识到当前的女娲一致性服务组不再继续工作，因此会主动退出各自的 FOLLOWING 状态，触发女娲一致性服务组新一轮主服务器选举，从而产生新的主服务器。

本故障是女娲一致性服务组主服务器的 Partial Failure 状态下运行多时

最终导致的故障，对应的解决问题的办法切换 LL 主服务器。因此能够根解本故障的“Fail-Stop Failure”即为女娲一致性服务组主服务器主动退出女娲一致性服务组，以触发新一轮主服务器选举。

（3）“Convert Partial Failure to Fail-Stop Failure”行为触发条件

女娲一致性服务组基于从服务器与主服务器之间定期心跳机制来判断整个女娲一致性服务组是否工作正常，在整个故障发生前后。ECS 集群里的女娲一致性服务组内从服务器与主服务器之间心跳正常，因此一直没有能够触发新的女娲一致性服务组新一轮主服务器选举。这个反映了不仅是女娲，还有很多应用服务的架构设计中的一个缺陷：心跳与服务脱节，即心跳除了反应进程状态，还应该携带更多服务状态信息。事实上，心跳是架构设计中非常普遍的一个机制，一般用于探测合作方是否仍然可以服务并交换一些关键信息。在不同的应用场景中，比如 LVS、无线、主备系统、多主系统，心跳设计如何准确且尽可能减少对资源的占用，这个是一个很大的话题，不再延伸讨论。

就解决本故障来说，针对女娲一致性服务组主服务器，技术人员引入待落盘事务性请求阻塞队列大小阈值，以及上次事务性请求落盘的时间与比当前时间距离的阈值，如果这两个阈值均达到女娲一致性服务组主服务器判断出现了数据落盘 Hang 住的“Partial Failure”，通过关闭与女娲一致性服务组中其余从服务的连接，退出自己 LEADING 状态，触发女娲一致性服务组中重新选举这一标准的“Fail-Stop Failure”。

小结

女娲，作为阿里云飞天系统中提供分布式协同服务的关键模块，它的使命是提供高效、高容错、强一致的分布式协同服务。特别是在今天阿里云如此规模的云计算场景下，一定会面临诸多挑战。

在面临未曾遇到的系统边缘场景的时候，我们能够严谨调查，追根溯源，并且做到跳出具体问题，从整体反思、解决问题，那么一定可以锤炼出更加稳定的系统。“路漫漫其修远兮，吾将上下而求索”，与诸君共勉。

4.4 疑案追踪，JSON 序列化不一致

背景

今天是圣诞节，开发小成早已周密安排好了和女友的约会。奈何人算不如天算，临下班的时候，一阵电话声拖住了小成回家的脚步。

“线上 app 功能有问题。”

开发小成：“我们排查一下，会尽快解决。”

没想到拉开了一场持续 27 小时的攻坚战。

原因定位

（1）尝试预发环境重现。

线上是集群服务，机器众多，直接定位困难，于是开发人员用测试账号尝试在预发环境重现问题。但是，在预发问题并不能重现，属于非必现的问题。

（2）准确定位出错机器，获取完整日志。

回到线上环境，直面庞大的集群，尝试重现问题并获取有效的错误日志。点击调试版 app 并不断人肉请求结果中的日志，终于从中找到几个有

效的业务跟踪日志采样。根据日志链路分析，请求从 app 处理网关到后端业务处理应用很快就结束返回了，订单提交处理没调用到任何业务相关的服务。业务跟踪日志指明了实际机器调用链路，直接登上机器拿具体的错误日志。尝试登录了两台机器，都拿到相同的日志信息：

```
   20xx-xx-xx 17:25:13.869 | 0a*186d8039c1417m |0.2| ERROR
c.a.*.TradeOrderServiceImpl - doCreatingOrderBiz error.
   com.alibaba.*.exception.BizFramworkRuntimeException:
ErrorMessage{rc='SYSTEM_ERROR',m='system error, might be npe.'}
      at sun.reflect.DelegatingMethodAccessorImpl.
invoke(DelegatingMethodAccessorImpl.java:43)
      at java.lang.reflect.Method.invoke(Method.java 498)
      at org.springframework.aop.support.AopUtils.
invokeJoinpointUsingReflection(AopUtil.java317)
      at org.springframework.aop.framework.
ReflectiveMethodInvocation.invokeJoinpiont(ReflectiveMethodInvocation.jav
a:183)
```

这段异常栈信息对应代码无法判定问题根本原因，大概推断是业务应用框架抛出来的一个未知异常，暂时判断与业务逻辑处理无关，继续追查。

（3）继续深究，获取完整线索。

此时拿到的错误信息不足以判定问题根本原因，开发人员随即向业务应用运维人员求助。由于业务应用对日志异常栈打印有行数限制控制，需要调整线上机器参数来放宽限制再获取更详细的日志。通过逐步放大行数限制，拿到完整有效的日志如下。

```
   20xx-xx-xx 17:25:13.869 | 0a*186d8039c1417m |0.2| ERROR
c.a.*.TradeOrderServiceImpl - doCreatingOrderBiz error.
   com.alibaba.*.exception.BizFramworkRuntimeException:
ErrorMessage{rc='SYSTEM_ERROR',m='system error, might be npe.'}
```

```
        ... …
        // *** 异常栈信息 ***
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java 498)
        at org.springframework.aop.support.AopUtils.
invokeJoinpointUsingReflection(AopUtil.java317)
        at org.springframework.aop.framework.
ReflectiveMethodInvocation.invokeJoinpiont(ReflectiveMethodInvocation.jav
a:183)
        … …
        at org.springframework.aop.framework.
JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204)
    Caused by: com.alibaba.fastjson.JSONException: syntax error, except {,
auctual false, pos 184, fieldName hidden
        at com.alibaba.fastjson.parser.deserializer.
JavaBeanDeserializer.deserialze(JavaBeanDeserializer.java:232)
        at com.alibaba.fastjson.parser.deserializer.
JavaBeanDeserializer.deserialze(JavaBeanDeserializer.java:135)
        at com.alibaba.fastjson.parser.deserializer.
ASMJavaBeanDeserializer.deserialze(ASMJavaBeanDeserializer.java:29)
        at com.alibaba.fastjson.parser.deserializer.
DefaultFieldDeserializer.parseField(DefaultFieldDeserializer:33)
        at com.alibaba.fastjson.parser.deserializer.
JavaBeanDeserializer.parseField(JavaBeanDeserializer.java:420)
        at com.alibaba.fastjson.parser.deserializer.
JavaBeanDeserializer.deserialze(JavaBeanDeserializer.java:330)
        at com.alibaba.fastjson.parser.deserializer.
JavaBeanDeserializer.deserialze(JavaBeanDeserializer.java:135)
        at com.alibaba.fastjson.parser.DefaultJSONParser.
parseObject(DefaultJSONParser.java:551)
```

```
    at com.alibaba.fastjson.JSON.parseObject(JSON.java:250)
    at com.alibaba.fastjson.JSON.parseObject(JSON.java:226)
    at com.alibaba.fastjson.JSON.parseObject(JSON.java:180)
    at com.alibaba.fastjson.JSON.parseObject(JSON.java:180)
    at com.alibaba.*.*BizOrderInitExt.
initOrderLineGroupWhenCreating(*BizOrderInitExt.java:132)
    ... …
```

终于，问题原因出现，是 app 提交的数据反序列化时报错，立即去截取 app 提交的数据样本，直接服务器暴力查找：

```
grep '0a*186d8039c1417m ' *.log
```

在业务应用参数提交日志里记录了完整的体检参数数据体，从中抽取出与报错对应的 JSON 数据体用作分析，此处省略具体的数据体内容。

根据日志信息也直接找到了准确的报错代码：

```
BizDataObj obj = JSON.parseObject(bizDataObj, new
TypeReference<com.alibaba.*.BizDataObj>() {});
```

代码也很简单，就是一个 JSON 格式字符串反序列化为 Java 对象的调用。

细节分析

（1）参数数据对比

找到了准确报错代码位置，缩小范围来看 app 提交的 JSON 是否正常，分别找一条正常的 JSON 请求参数数据和报错的 JSON 请求参数数据做对比。

正常的 JSON 请求参数数据片段：

```
"bizDataObj_json": {
    ... …
    "id": "a186d8039c1417m",
    "hidden": {
        "empty": true,
        "notEmpty": false
    }
    ... …
}
```

报错的 JSON 请求参数数据片段：

```
"bizDataObj_json": {
    ... …
    "id": "a186d8039c1417m",
    "hidden": false
    ... …
}
```

根据错误日志“syntax error, expect {, actual false, pos 184, fieldName hidden”，就是 JSON 数据中的 hidden 属性有问题，直接对比这个属性的字符串。

正常参数中的 hidden 属性：

```
"hidden": {
    "empty": true,
"notEmpty": false
}
```

报错参数数据中的 hidden 属性：

```
"hidden": false
```

很明显，正常参数数据中的 hidden 属性是被当作 Object 类型序列化的，而报错参数中的 hidden 属性是被当作布尔（Boolean）原生类型序列化了。而参数提交做反序列化处理时期望的是对象类型，此时提交上来的包含布尔（Boolean）类型 hidden 属性的 JSON 参数串就导致了异常报错。

（2）进一步排查

线上怎么会出现两种序列化的结果？从出错根本原因来推断，真正的问题在 JSON 序列化时产生的，所以先看这个 hidden 属性从哪里来，找到对应的类定义代码。

被序列化为 JSON 的类代码：

```
public class BizDataObj extends BasicCO {
    public static final String tag = "bizDataObj";
    private Long id;
        private Long userid;
        ... …
        // 属性 get 和 set 方法列表
}
```

他山之石可以攻错，BizDataObj 是一个业务组件类，报错的那段业务处理使用了这个组件类来达到业务传值的目的。而我们关注的 hidden 属性并没有在这定义，它在 BasicCO 那，查看父类代码。

同时也被序列化的父类代码：

```
public abstract class BasicCO implements Component, Serializable {
    private static final long serialVersionUID = 1L;
    ... ...
    public Hidden hidden;
```

```
    ... ...
    public Hidden getHidden() {
        return hidden;
    }

    public boolean isHidden() {
        return this.getStatus();
    }
}
```

BasicCO 是被序列化的 BizDataObj 的父类，引起问题的 hidden 属性就定义在这里。这个属性的 Getter 方法的同时有 getHidden()和 isHidden()两种形式，这会导致反序列化反射调用时可能出现随机性结果，不同机器可能调用到的具体 Getter 方法不一样。

当时线上业务代码采用的 JSON 采用序列化工具是 fastjson-1.2.4，此版本调用 java.lang.Class.getMethods 方法来获取 JavaBean 的 Getters 来做反序列化，无法保证有重复 getters 的序列化调用一致性，所以线上随机性出现两种序列化结果，导致用户不规律出现提交订单出错。

解决方案

（1）升级 Fastjson 二方库——问题更严重

JavaBean 序列化方法加载优先级在新版的 Fastjson 有解决，那么我们做出了升级 Fastjson 版本到 1.2.19 的决定。然而升级完成后问题更严重了，本来概率性出现的问题变成了 100%出现。

前一个问题解决却又引发了新的问题，反序列化时对象初始化方法调用又抛出异常，又查 Hidden 类的定义，它的构造函数是私有的：

```
public class Hidden implements IHidden {
    ... ...
    private Hidden() {
    }

    public static Hidden of() {
        return new Hidden();
    }
    ... ...
}
```

Fastjson-1.2.19 的版本对私有构造函数的对象不能正常初始化，尽管 Fastjson 的作者紧急发布了 1.2.20 版本，但是开发人员不敢贸然再次直接升级三方库。Fastjson 在业务应用的其他很多地方也有引用，新版本的贸然引入还有可能会引发其他问题，风险不可控。如果再次做基础工具库升级，需要做全面的测试验证，而此时做全面业务验证的压力太大，短时间难以完成。此时需要的是能收敛问题影响、简单有效的解决方案。

（2）呆板的方法

hidden 属性是被继承引入但在业务处理中不使用，所以直接删除可以让业务代码往下走而又不影响其他业务。于是在 JSON 反序列化之前强制把 hidden 属性删除可以使得 Fastjson 反序列化时忽略 hidden 属性从而避开问题。此方法看上去比较低端，但是这个方案只改动问题本代码，不会影响其他业务实现，可以快速有效地解决问题并控制风险，实现代码如下：

```
// fixme 临时修复 强制 hidden 属性忽略，解决 hidden 属性 get 方法重复的潜在缺陷
Map propMap = JSON.parseObject(ctRelated);
propMap.remove("hidden");
String tmpJsonStr = JSON.toJSONString(propMap);
BizDataObj obj = JSON.parseObject(tmpJsonStr, BizDataObj.class);
```

这里产生了一个疑问：这个 BasicCO 组件是个基础组件，别的业务也在引用，为什么只这个场景的业务出问题？

原因是其他业务实现的组件序列化用的是 Jackson，而我们在业务代码里用 Fastjson 来序列化。Jackson 和 Fastjson 属于两种 Java 对象 JSON 解析器的实现。Fastjson 要求被序列化的类符合 JavaBean 规范，Jackson 对被序列化对象没有特殊要求(POJO 即可)。Jackson 可以较为灵活地指定 JSON 解析策略，业务实现中使用基于对象 field 的解析化方案，可以避免 JavaBean Getter 方法不唯一带来的序列化结果不一致问题。

小结

（1）第三方代码引入保持隔离性

在业务处理中直接引入一个二方库组件类（即 BizDataObj ）对象并对其做序列化，而出问题的地方却是被这个类继承的 BasicCO 基类。在业务场景中基类 BasicCO 的属性其实都不使用，更好的代码实现模式应该是在业务实现中按业务要求定义一个类来专供业务处理，与第三方类之间可以使用适配器模式 Adapter 保持联系与隔离，不完全了解而引入的类直接使用，可能带来风险。引申开来，我们在引入第三方的代码时应该有隔离层，在隔离层中定义需要明确使用的业务功能，通过设计模式将第三方代码提供的实现隔在外，避免第三方代码缺陷或者多余功能特性影响到自己的代码。

（2）关键日志记录

面对应用服务集群化运维的时代，像本案例中这类非必现问题，全链路业务跟踪日志的记录手段对于问题的快速定位尤其重要；本案例中 app 的日志机制和业务应用的关键参数日志记录都比较完整，而且实现了全链路透传的业务跟踪日志，所以问题原因定位过程有据可循；同时也需要考

虑在高访问量的应用系统中日志过多也可能成为负担，要在全面信息和有效信息的记录之间取得平衡。

（3）紧急修复方案更加关注准确性和高效性

本案例中问题的解决过程并不顺利，找到问题后由修复一个问题而引发另一个问题，有问题本身的特殊性，也有方案选择上可以思考的地方。对于线上问题，止血是第一要务，尽量用影响面小的方案，问题的解决方法尽量收敛到一个点，避免采用针对面的解决方案，完美代码和风险控制之间需要有取舍。

（4）代码规范性检查

本案例中 JavaBean 的 Getter 方法定义出现了二义的代码写法也是引发问题的关键因素，代码书写规范性永远需要重视，开发流程中可以引入代码规范性检查工具（比如 CheckSytle）代码缺陷分析工具（比如 FindBugs）等开发检查工具来帮助规避潜在问题。

4.5　从现象到本质，不保证顺序的 Class.getMethods JVM 实现

背景

有个线上系统踩了一个坑，技术人员定位到是 java.lang.Class.getMethods 返回的顺序不同机器可能不一样，有问题的机器和没问题的机器返回的方法列表是不一样的，后面系统相关的人来找笔者求证 JDK 里是否有这样的潜规则。

这个问题本来很简单一句话就可以说明白，不过可能很多人会问以下两个问题。

- 为什么有代码需要依赖这个顺序？
- JVM 里为什么不保证顺序？

本文主要就针对这两个问题展开说一下。

（1）依赖顺序的场景

如果大家看过或者实现过序列化反序列化的代码，这个问题就不难回答了，今天碰到的这个问题其实是发生在大家可能最常用的 Fastjson 库里的，所以如果大家在使用这个库，请务必检查下你的代码，以免踩到此坑。

（2）对象序列化

大家都知道当我们序列化好一个对象之后，要反序列回来，那问题就来了，就以这个 JSON 序列化为例，我们要将对象序列化成 JSON 串，意味着要先取出这个对象的属性，然后写成键值对的形式，那取值就意味着我们要遵循 JavaBean 的规范通过 getter 方法来取。其实 getter 方法有两种，一种是布尔（Boolean）类型的，一种是其他类型的。如果是布尔（Boolean）类型的，我们通常是 isXXX()这样的方法，如果是其他类型的，一般是 getXXX()这样的方法。

假如说我们的类里针对某个属性 a，同时存在两个方法 isA()和 getA()，究竟会调用哪个来取值？这个就取决于具体的序列化框架实现了，比如导致该案例诞生的 Fastjson，就是利用本文的主角 java.lang.Class.getMethods 返回的数组，然后挨个遍历，先找到哪个就是哪个。如果这个数组正好因为 JVM 本身实现没有保证顺序，那么可能先找到 isA()，也可能先找到 getA()，如果两个方法都是返回 a 这个属性其实问题也不大，但是假如这两个方法返回了不同的内容呢？

```
private A a;
public A getA(){
    return a;
}
public boolean isA(){
    return false;
}
public void setA(A a){
    this.a=a;
}
```

如果是上面的内容，选了 isA()，那其实是返回一个布尔（Boolean）类型的，将这个布尔（Boolean）写入到 JSON 串里；如果是选了 getA()，就是将 A 这个类型的对象写到 JSON 串里。

（3）对象反序列化

在完成了序列化过程之后，需要将这个字符串进行反序列化了，于是就会去找 JSON 串里对应字段的 setter 方法，比如上面的 setA(A a)。假如我们之前选了 isA()序列化好内容，那此时的值是一个布尔（Boolean）值 false，就无法通过 setA 来赋值还原对象了。

解决方案

相信大家看完上面的描述，知道这个问题所在了，要避免类似的问题，方案较多，比如对方法进行先排序，或者说优先使用 isXXX()方法，不过需要和开发者达成共识，和 setter 要对应起来。

JVM 里为什么不保证顺序

JDK 层面的代码本文暂且不提，大家都能看到代码，从

java.lang.Class.getMethods 一层层走下去，相信读者细心点还是能抓住整个脉络的，这里主要想说大家可能比较难看到的一些实现，比如 JVM 里的具体实现。

正常情况下大家跟代码能跟到调用 java.lang.Class.getDeclaredMethods0 这个 native 方法，其具体实现如下。

```
JVM_ENTRY(jobjectArray, JVM_GetClassDeclaredMethods(JNIEnv *env, jclass
ofClass, jboolean publicOnly))
  {
    JVMWrapper("JVM_GetClassDeclaredMethods");
    return get_class_declared_methods_helper(env, ofClass, publicOnly,
                                             false,
SystemDictionary::reflect_Method_klass(), THREAD);
  }
  JVM_END
  其主要调用了 get_class_declared_methods_helper 方法
  static jobjectArray get_class_declared_methods_helper(
                                JNIEnv *env,
                                jclass ofClass, jboolean publicOnly,
                                bool want_constructor,
                                Klass* klass, TRAPS) {
    JvmtiVMObjectAllocEventCollector oam;
    if (java_lang_Class::is_primitive(JNIHandles::resolve_non_null
(ofClass))
        || java_lang_Class::as_Klass(JNIHandles::resolve_non_null
(ofClass))->oop_is_array()) {
      oop res = oopFactory::new_objArray(klass, 0, CHECK_NULL);
      return (jobjectArray) JNIHandles::make_local(env, res);
    }
    instanceKlassHandle k(THREAD,
java_lang_Class::as_Klass(JNIHandles::resolve_non_null(ofClass)));
```

```
    k->link_class(CHECK_NULL);
    Array<Method*>* methods = k->methods();
    int methods_length = methods->length();
    ResourceMark rm(THREAD);
    GrowableArray<int>* idnums = new GrowableArray<int>(methods_length);
    int num_methods = 0;
    for (int i = 0; i < methods_length; i++) {
      methodHandle method(THREAD, methods->at(i));
      if (select_method(method, want_constructor)) {
        if (!publicOnly || method->is_public()) {
          idnums->push(method->method_idnum());
          ++num_methods;
        }
      }
    }
    objArrayOop r = oopFactory::new_objArray(klass, num_methods,
CHECK_NULL);
    objArrayHandle result (THREAD, r);

    for (int i = 0; i < num_methods; i++) {
      methodHandle method(THREAD, k->method_with_idnum(idnums->at(i)));
      if (method.is_null()) {
        result->obj_at_put(i, NULL);
      } else {
        oop m;
        if (want_constructor) {
          m = Reflection::new_constructor(method, CHECK_NULL);
        } else {
          m = Reflection::new_method(method, UseNewReflection, false,
CHECK_NULL);
        }
        result->obj_at_put(i, m);
```

```
    }
  }
  return (jobjectArray) JNIHandles::make_local(env, result());
}
```

从上面的 k->method_with_idnum(idnums->at(i))，基本知道方法主要是从 klass 里来的。

```
Method* InstanceKlass::method_with_idnum(int idnum) {
  Method* m = NULL;
  if (idnum < methods()->length()) {
    m = methods()->at(idnum);
  }
  if (m == NULL || m->method_idnum() != idnum) {
    for (int index = 0; index < methods()->length(); ++index) {
      m = methods()->at(index);
      if (m->method_idnum() == idnum) {
        return m;
      }
    }
  return NULL;
  }
  return m;
}
```

因此 InstanceKlass 里的 methods 是关键，而这个 methods 的创建是在类解析的时候发生的。

```
instanceKlassHandle ClassFileParser::parseClassFile(Symbol* name,
                                                    ClassLoaderData* loader_data,
                                                    Handle protection_domain,
                                                    KlassHandle host_klass,
                                                    GrowableArray<Handle>* cp_patches,
```

```
                                TempNewSymbol& parsed_name,
                                               bool verify,
                                               TRAPS) {
  ...
   Array<Method*>* methods = parse_methods(access_flags.is_interface(),
                                        &promoted_flags,
                                        &has_final_method,
                                    &declares_default_methods,
  ...                                    CHECK_(nullHandle));
  // 排序方法
  intArray* method_ordering = sort_methods(methods);
  ...
  this_klass->set_methods(_methods);
  ...
  }
```

上面的 parse_methods 就是从 class 文件里挨个解析出 method，并存到 _methods 字段里，但是接下来做了一次 sort_methods 的动作，这个动作会对解析出来的方法做排序。

```
  intArray* ClassFileParser::sort_methods(Array<Method*>* methods) {
    int length = methods->length();
    if (JvmtiExport::can_maintain_original_method_order() ||
DumpSharedSpaces) {
      for (int index = 0; index < length; index++) {
        Method* m = methods->at(index);
        assert(!m->valid_vtable_index(), "vtable index should not be set");
        m->set_vtable_index(index);
      }
    }
    Method::sort_methods(methods);
    intArray* method_ordering = NULL;
    if (JvmtiExport::can_maintain_original_method_order() ||
```

```
DumpSharedSpaces) {
      method_ordering = new intArray(length);
      for (int index = 0; index < length; index++) {
        Method* m = methods->at(index);
        int old_index = m->vtable_index();
        assert(old_index >= 0 && old_index < length, "invalid method index");
        method_ordering->at_put(index, old_index);
        m->set_vtable_index(Method::invalid_vtable_index);
      }
    }
    return method_ordering;
  }

  void Method::sort_methods(Array<Method*>* methods, bool idempotent, bool
set_idnums) {
    int length = methods->length();
    if (length > 1) {
      {
        No_Safepoint_Verifier nsv;
        QuickSort::sort<Method*>(methods->data(), length,
method_comparator, idempotent);
      }
      // 重置方法顺序
      if (set_idnums) {
        for (int i = 0; i < length; i++) {
          Method* m = methods->at(i);
          m->set_method_idnum(i);
          m->set_orig_method_idnum(i);
        }
      }
    }
  }
```

从上面的 Method::sort_methods 可以看出其实具体的排序算法是 method_comparator。

```
static int method_comparator(Method* a, Method* b) {
 return a->name()->fast_compare(b->name());
}
```

比较的是两个方法的名字，但是这个名字不是一个字符串，而是一个 Symbol 对象。每个类或者方法名字都会对应一个 Symbol 对象，在这个名字第一次使用的时候构建，并且不是在 java heap 里分配的，比如 JDK 7 里就是在 cheap 里通过 malloc 来分配的，JDK 8 里会在 metaspace 里分配。

```
int Symbol::fast_compare(Symbol* other) const {
 return (((uintptr_t)this < (uintptr_t)other) ? -1
  : ((uintptr_t)this == (uintptr_t) other) ? 0 : 1);
}
```

从上面的 fast_compare 方法可以知道，其实对比的是地址的大小，因为 Symbol 对象是通过 malloc 来分配的，因此新分配的 Symbol 对象的地址不一定比后分配的 Symbol 对象地址小，也不一定大，因为期间存在内存 free 的动作，那地址不是线性变化的，之所以不按照字母排序，主要还是为了速度考虑，根据地址排序是最快的。

综上所述，一个类里的方法经过排序之后，顺序可能会不一样，取决于方法名对应的 Symbol 对象的地址的先后顺序。

JVM 为什么要对方法排序

其实这个问题很简单，就是为了快速找到方法。当我们要找某个名字方法的时候，根据对应的 Symbol 对象，能根据对象的地址使用二分排序的算法快速定位到具体的方法。

小结

当我们在对技术细节了解不够深入的时候，尽量不要建立在假设的前提下来实现我们的逻辑。比如本文里提到的场景，我们假定 JDK 里这块的实现顺序是固定的，其实不然。大家要小心处理，能自己控制的逻辑就主要自己控制。

4.6 破解超时迷局，浅析启动初期 load 飙高问题

背景

会员中心应用（以下简称 MC），为上层业务提供了会员信息的读写服务。其应用调用存在几个特点：调用方多、高 QPS 低 RT、请求集中在少数接口。

MC 在发布过程中，会有少量请求超时情况，发布期间会有少数机器 load 飙高继而报警，且每次报警的机器不同。

某个夜深人静的晚上，开发人员准备发布一个业务需求，发布一段时间后 MC 请求处理出现超时，上层业务的调用纷纷超时，几分钟内自动恢复。

原因定位

（1）查询调用链路图

首先如图 4-11 所示，查询超时请求调用链路图具体如下，发现 MC 应用发起的请求，服务端处理耗时 1ms，MC 收到应答耗时 4s+，怀疑是 MC

主机网络异常或 load 过高引起，下一步查看请求所在 MC 主机监控数据。

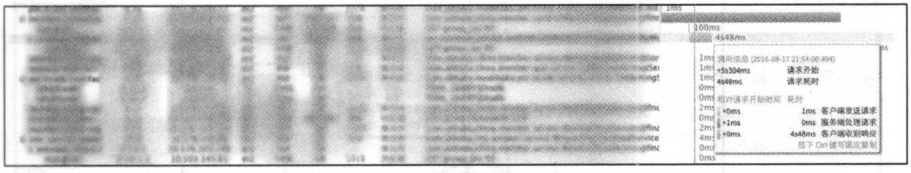


图 4-11　查询超时请求调用链路

（2）查询所在主机系统资源使用曲线图

查看监控系统中 MC 主机监控数据，发现网络无异常，无 FGC，而 CPU 和 load 飙高。说明 MC 收到应答耗时太长是因 MC 主机 load 升高引起，接下来对比发布过程中超时请求所在主机的 load 情况。如图 4-12 和图 4-13 所示。

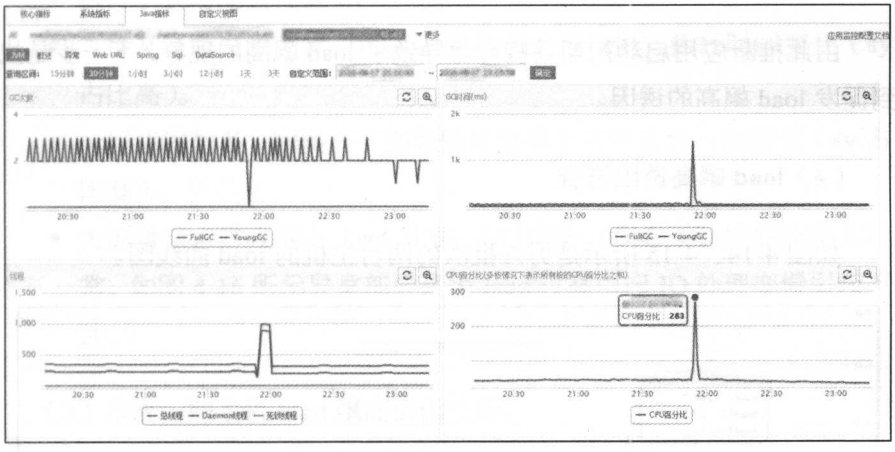


图 4-12　主机 1 的 load 曲线图

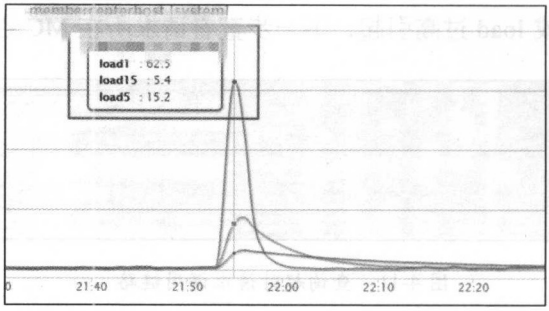


图 4-13　主机 2 的 load 曲线图

（3）对比其他问题主机印证猜测

查看了发布阶段的其他超时请求，发现对应主机也存在 load 飙高的情况，发生的时间都在应用启动后 1 分钟之内。此现象印证了请求超时是因启动初期 load 飙高引起。

由此推断应用启动初期某些行为导致了 load 飙高的现象，下一步是找到促发 load 飙高的诱因。

（4）load 飙高诱因排查

如图 4-14、4-15 所示是另外批次的两台主机的 load 曲线图。

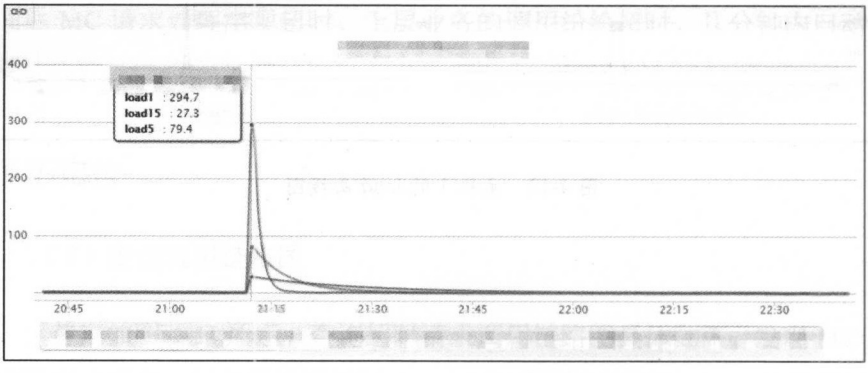


图 4-14　主机 3 的 load 曲线图

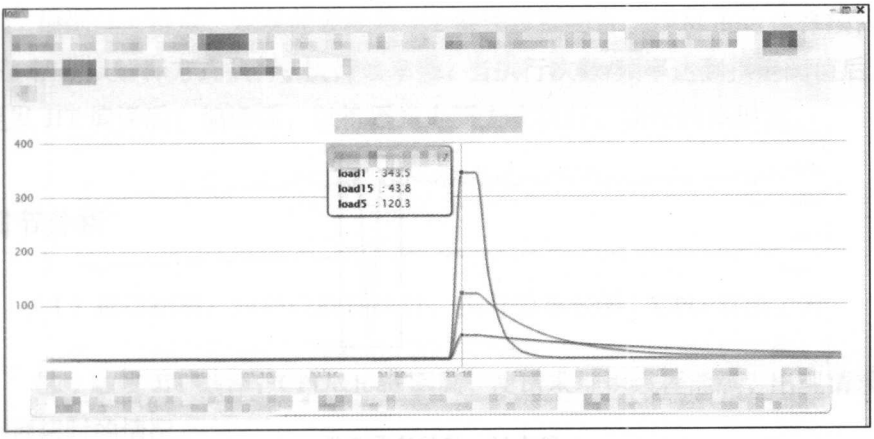


图 4-15　主机 4 的 load 曲线图

（1）load 升高一般可能存在的原因如下。

- 执行了长时间且耗费 CPU 的操作（us 占比高）。
- 等待某竞争资源（锁、IO）致使请求处理缓慢，执行时间变长（sy 占比高）。
- 运行线程数多，切换上下文频繁导致单个任务处理时间变长（sy 占比高）。
- 大流量请求进入导入 load 升高。（已排除，查看了监控的流量无异常，如图 4-16 所示异常部分是因 load 过高引起 IO 处理变慢引起下跌。）

（2）本次问题主机 load 飙高诱因分析。

结合日志发现问题 MC 主机先发生 CPU 升高，后出现 YGC 时间升高，最后 HSF 线程池满，异常出现。

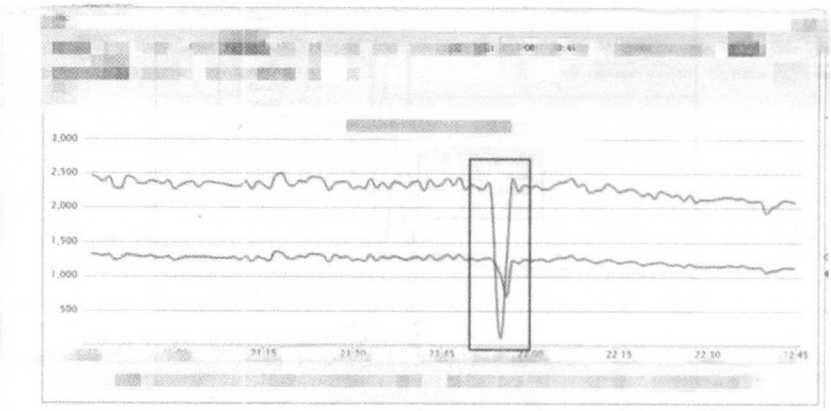


图 4-16 监控流量曲线

结合上述 load 升高的一般诱因，结合 MC 应用的特点，排除资源竞争和请求突增的情况；应是启动初期执行了长时间且耗费 CPU 的操作，然后 load 升高，YGC 时间升高，HSF 线程池满。下一步排查导致 CPU 升高的操作。

（3）CPU 占用升高原因排查：

排查导致 CPU 升高的操作，从三方面考虑：一类是业务代码执行；一类是框架&中间件初始化过程；第三类是容器方面（jvm\web 容器）。

首先排查 MC 应用有无业务请求耗费 CPU 且执行时间较长，排查下来，发现无此类业务请求。

然后分析了问题主机日志&应用启动过程，未发现可疑点。

然后分析下问题主机的表现如下。

- load 飙高出现在启动初期；
- load 飙高现象偶发（非固定主机）；
- load 飙高持续 1~2 分钟后恢复。

根据以上特点，发现此表现与 JIT 编译过程吻合。JVM 混合执行模式下，初期以解释方式执行，执行效率慢；当执行次数/频率达到指定阀值后，促发 JIT 编译后；编译后，以机器指令码方式执行，执行效率提高。

细节分析

（1）启动初期：应用以解释执行，执行效率较慢，CPU 有所上升；

（2）触发 JIT 编译：CPU&load 飙高，使请求堆积线程池满，出现请求处理超时的情况；

（3）JIT 编译后：CPU 降低，请求处理较之前快，服务逐步恢复。

解决方案思考

针对上述问题，有以下几种解决方案。

（1）分层发布——降低请求流量

实现方法：启动初期，降低新启动 provider 的权重（减少流量），指定时间后恢复为正常。

劣势：权重参数不可调整，不能确保 JIT 完成前不引发超时。

优势：与业务无关，无须代码植入。

结论：不选用，因不可用，MC 做过之后，仍出现服务端超时情况。

（2）升级 JDK8——缓存编译结果

实现方法：升级 JDK8，开启热地代码日志功能，JVM 退出时将 HOTCACHE 缓存到本地，下次启动时，载入上次缓存的 HOTCACHE。

劣势：升级 JDK8，影响较大，需升级编译环境，部署环境，并要做完整测试保证。

优势：能够将执行期间优化的编译结果缓存，并直接载入使用，无须再次运行期优化过程，与业务无关，无须代码植入。

结论：暂不选用，因替换成本太大。

（3）容器启动前预热触发 JIT 编译——提前促发编译

实现方法：在容器启动前，循环调用热点方法，触发 JIT 编译，而后容器启动完成，对外发布 HSF 服务。

劣势：需代码植入，增加启动时间。

优势：实现成本低，不影响业务。

结论：选用。

具体实施方案

（1）CodeWarmupContextListener 实现 ApplicationContextAware 接口，确保在容器启动完成前，调用预热方法；

（2）CodeWarmupContextListener 读取预热配置（目标方法、请求参数、执行次数和超时时间）；

（3）采用线程池方式执行目标方法，采用线程池方式，泛化调用 spring BeanFactory 中的目标方法，执行 *N* 次以促发 JIT 编译；

（4）执行完成，调用线程池关闭；

（5）容器启动完成；

（6）对外发布 HSF 服务。

小结

业界 JVM 多使用混合执行模式，其执行过程按照如下步骤。

（1）解释执行：在应用启动之后，方法以解释执行方式执行。

（2）触发 JIT 编译：当方法执行次数达到 JIT 编译阈值时（server 方式下默认为 10000 次，计数会根据执行时长做衰减，以防对非热点方法执行 JIT 编译）会触发 JIT 编译。

（3）机器指令方式执行：编译之后，此方法以机器指令方式执行，进而提高执行效率（Tiered compilation，在编译以后的执行期间，JVM 还会根据执行情况对编译后的指令做优化处理，进而更好地提高执行效率。此处不展开）。

此种方式本没有问题，可是当应用调用量很大时，大量请求同事进入，多个方法同时促发 JIT 时，会出现本文中上述提到的启动初期 load 飙高，进而导致服务超时的问题。因此问题出现有随机性，问题的持续时间与应用主机当时的负载有关系，排查和重现有一定难度。

如需紧急解决发布初期 load 飙高的问题且热点方法比较集中，可以使用本文中提到的方法（3）。但从长期来看，还是建议大家升级 JDK8 使用 hotcache 解决此问题。

4.7 洞悉千丝万缕，浅谈 JIT 编译优化的误区

背景

某天下午，开发人员对业务系统 H 按常规做了一次线上发布，在发布过程中收到大量接口调用超时的报警，排查发现发布过程中大量已发布机器 CPU 使用率接近 100%，并伴随系统 1 分钟 load 飙高到 100 以上。

```
---cpu-- ---mem-- ---tcp-- -----traffic----    --sda---  ---load-
util    util    retran   bytin    bytout   util     load1
29.34   50.04   0.14     42.4K    37.2K    5.66     1.21
51.66   57.50   0.07     256.5K   185.5K   1.93     2.06
81.50   12.93   0.05     4.4M     3.2M     1.56     93.08
93.37   61.70   0.07     3.8M     2.3M     1.43     38.51
98.68   63.86   0.07     4.6M     3.4M     1.64     91.93
99.49   64.52   0.09     5.0M     3.5M     1.26     131.10
```

原因定位

第一阶段，为了避免大接口调用超时影响线上业务的稳定性，立刻回滚业务代码，并重启应用，但是代码回滚后，机器 CPU 和 load 在重启过程中依然飙高，由此断定系统负载变高与代码发布无关。为了找出 CPU 飙高的原因，开发在重启过程中通过执行以下命令找出 CPU 利用率占比较高的线程。

```
ps -eL -o pid,%cpu,lwp|grep -i pid | sort -nr -k2 | awk
'{printf("%s %s %x\n",1,1,2,$3)}'|head
```

配合 Jstack Dump 的线程栈，发现是 JIT 过程中 C2 的两个 CompilerThread 占用大量 CPU。

```
   "C2 CompilerThread1" daemon prio=10 tid=0x00007f4cbc146800 nid=0x33644
runnable [0x0000000000000000] java.lang.Thread.State: RUNNABLE
   "C2 CompilerThread0" daemon prio=10 tid=0x00007f4cbc144800 nid=0x33643
runnable [0x0000000000000000] java.lang.Thread.State: RUNNABLE
```

至此，定位出系统发布时 CPU 和 load 飙高的原因是发布过程中应用重启，JAVA 应用需要通过 JIT 将字节转码转换成 Native Code 提升应用性能，刚启动时以上工作尚未完成，应用的性能较差，JIT 本身还占用了大量 CPU 导致应用的负载较高，出现接口调用超时等现象。

原因明确后，我们知道在 JDK7 里可以通过增加 JVM 参数 -XX:+TieredCompilation，将 JIT 的编译方式改为分层编译缓解此问题，并在个别机器上验证可行后，全量修改业务系统 H 所有的系统启动脚本，并重新发布之前的业务代码。

第二阶段，业务系统发布完成 1 小时之后，出现业务异常报警。我们通过监控发现，此时系统存在大量的接口调用超时，所有接口的响应时间非常高，而此时系统的负载并不高。由于之前刚对系统做过发布，怀疑是变更导致，因此收到报警的第一时间，先做了代码回滚。回滚完成后，业务有短暂恢复，但随即又出现异常，之后，相继重启业务系统的所有机器，以及将业务流量切到其他单元（该系统采用了异地多活的架构，实现了多地区多机房部署，机房间的流量可以秒切，当某个机房出现故障时可以将流量切到其他机房，实现故障的瞬间恢复），但业务都只是短暂恢复。

第三阶段，此时我们通过性能分析工具 Perf 发现系统存在明显的与 CodeCache 相关的热点代码，根据以往经验初步怀疑是因为 CodeCache 被填满导致 JIT 失败，应用性能严重下降，JAVA 应用本身变慢，而分层编译

采用 C1 和 C2 两种编译方式，CodeCache 中会同时存在 C1 和 C2 编译过的代码，更容易将 CodeCache 填满，所以决定调大 CodeCache，并去掉分层编译，随即将 JVM 参数-XX:ReservedCodeCacheSize 调大到 256MB，并去掉-XX:+TieredCompilation，重启系统并观察了一段时间后，业务恢复并持续平稳，至此问题得到解决。

详细分析

为什么去掉 JVM 分层编译参数-XX:+TieredCompilation，以及调大 JVM 参数-XX:ReservedCodeCacheSize 可以解决问题？

首先，根据之前分析，增加 JVM 参数-XX:+TieredCompilation 可以让 JIT 同时使用 C1 和 C2 两种编译方式，提高 JAVA 将字节码转换为 Native Code 的效率，但却增加了对 CodeCache 空间的消耗，导致整个 CodeCache 被占满，应用性能出现急剧下降。CodeCache 被填满后会在应用日常打印“CodeCache is full. Compiler has been disabled”，但搜索所有机器日志都未发现相关记录，故怀疑当时 CodeCache 并未被填满，从 Perf 分析的结果上推断肯定是与 CodeCache 和设置-XX:+TieredCompilation 有关。

其次，利用一台机器对该问题进行重现，添加分层编译参数，重启应用，开启 Perf 查看热点代码，刚开始并没有发现特别的热点，并在系统运行过程中通过 JMX 查看 CodeCache 空间的大小（JDK1.7 JVM CodeCache 的默认大小是 96MB），观察发现在应用启动之后，CodeCache 的空间快速消耗，很快就达到即将占满的状态。当达到一个临界点之后 CodeCache 的增长开始放缓。十几分钟之后 CodeCache 的使用空间开始变小，说明触发了 JVM 对 CodeCache 中的内容进行了回收，后续 CodeCache 空间的变化不太明显，但也略有缩小。

```
Jsage = {
  committed = 99155968;
```

```
    init = 2555904;
    max = 100663296;
    used = 98109696;
  };

  Jsage = {
    committed = 99155968;
    init = 2555904;
    max = 100663296;
    used = 68528256;
  };
```

此时，Perf 中的热点代码也有所变化，出现了故障时间相同的热点代码——largest_free_block（如图 4-17 所示），通过压测工具发起压测，当机器 QPS 变大时，应用的响应时间集体飙高，与预期相符。

图 4-17　热点代码

最后，根据热点代码 largest_free_block，分析 JVM 源码里的 largest_free_block()函数。

```
size_t CodeCache :: largest_free_block(){
    if (CodeCache_lock->owned_by_self()){
        return _heap->largest_free_block();
    } else {
        MutexLockerEx mu(CodeCache_lock, Mutex ::
_no_safepoint_check_flag);
        return _heap->largest_free_block();
```

```
    }
}
```

从代码上看，largest_free_block 首先会持有一个锁，然后去调用到_heap->largest_free_block()，代码中的_heap 是 CodeHeap 对象，代码如下。

```
size_t CodeHeap :: largest_free_block() const {
  size_t free_sz = size(_free_segments);
  size_t unused = max_capacity() - allocated_capacity() - free_sz;
  if (unused >= free_sz)
    return unused;
  size_t len= 0;
  for (FreeBlock* b = _freelist; b != NULL; b = b->link()) {
    if (b->length() > len)
      len = b->length();
}
return MAX2(unused, size(len));
```

当 JVM 触发对 CodeCache 的回收时，freelist 会维护所有的 Code Cache 空闲空间的链表（如图 4-18 所示）。

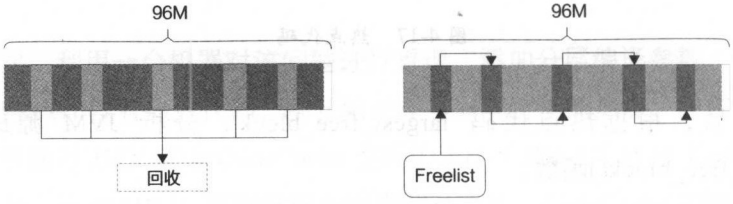


图 4-18 CodeCache 内存管理策略

当在进行 JIT 编译时，JVM 都会调用 largest_free_block()找到一个存放 Native 代码的空间，代码中的 free_sz 代表的是整个 freelist 包含的大小，unused 代表的是 CodeCache 剩余未分配的大小。当 CodeCache 剩余未分配的空间大于 freelist 所包含的空间时，直接从 CodeCache 剩余未分配空间中

进行分配，否则，会遍历所有的 freelist，找到其中最大的空间，并返回与 CodeCache 剩余未分配空间中最大的那个空间。由于分层编译会采用 C1 和 C2 两种编译方式，导致编译后的代码被频繁写入或失效 CodeCache，使得 freelist 变得非常碎片化，而编译后的代码通常又比较小（几百字节），按回收之后的空闲空间是 30MB 计算，那么这个 freelist 的长度将变得十分巨大。每次 JVM 进行 JIT 操作时，在持有锁的情况下去遍历这样一个巨大的 freelist 将非常的耗时，未被编译成 Native Code 的代码只能做解释执行，而此时 JIT 编译的效率已经严重下降，因此，在这种情况下会导致应用响应时间暴涨，线程池满，响应超时，通过代码分析发现 JVM 在分层编译情况下是存在严重 bug 缺陷的。

为什么处理过程中重启机器和切流只会带来短暂的业务恢复？

问题处理过程中，重启机器、单元间切流均是在应用开启了分层编译参数-XX:+TieredCompilation 的情况下操作的，CodeCache 的严重碎片化需要一定时间才会生产生，应用刚启动时碎片化还不严重，所以业务短时间内是正常的，一旦碎片化，应用整体的性能严重下降，系统响应时间暴涨。

解决方案

在 JDK 7 里不要使用-XX:+TieredCompilation 来缓解应用重启过程中，JIT 编译导致的 CPU 使用率达 100%的问题，可以通过提前代码预热或增加应用容量来解决。

设置 JVM 参数-XX:ReservedCodeCacheSize=256MB，防止应用出现因为 CodeCache 空间不足导致的问题。

小结

底层 JVM 参数的调整，尽量 Beta 的时间足够长进行充分验证，在正式发布之前最好通过压测发起大流量来验证是否带来性能问题。

故障出现时，所有的变更包括代码和配置都要一起回滚，最好将配置做到代码里，比如应用的 Docker 化。

第5章 运行管理域稳定性建设

阿里巴巴集团自创立时起，始终在提升系统稳定性领域进行积极实践。2004 年阿里巴巴集团成立了第一个监控中心团队，十年之内逐渐加强从故障发现到应急响应到故障防范各领域的理论和能力建设，并且于 2015 年成立了 GOC（Golbal Operations Center，全球运行指挥中心），建立了横向打通阿里集团所有业务单元故障全域的运行管理体系，简称运行管理域。

运行管理域包含故障应急体系和以故障/容灾演练、稳定性运营为主要手段的故障防范体系。其中每一个领域都有对应的服务支持和工具支撑。经历了线上化到自动化的过程至今，GOC 已经开始在故障自动定位、自动恢复等领域进行智能化运行管理手段的探索。

5.1 洞若观火，让故障无处遁形

背景

2016 财年，阿里巴巴电商交易额（GMV）突破 3 万亿元人民币，成为全球最大网上经济体，这背后离不开基础设施事业群构筑的坚强基石。全球运行指挥中心（GOC，Golbal Operations Center）隶属于阿里巴巴基础设施事业群，是阿里电商、蚂蚁、菜鸟、云、平台技术等集团各项业务线上运行的核心保障团队，肩负着让用户因故障造成的伤害“少一秒钟”的使命，扎根运行管理域，通过构筑涵盖故障管理、监控管理、变更管理、演练管理、活动管理、稳定性文化建设的运行管理体系，与各一线业务团队共同劈波斩浪、创造奇迹，提升业务竞争力。

在 2016 年双十一全球购物狂欢节中，天猫全天交易额 1207 亿元，前 30 分钟每秒交易峰值 17.5 万笔，每秒支付峰值 12 万笔。阿里巴巴经济体庞大的业务系统为什么稳如泰山，各类问题是如何以迅雷不及掩耳之势被解决，从而对客户毫无影响？本文将为你一一解开这些谜团。

阿里应急体系

如图 5-1 所示，是阿里持续运作多年的故障体系，GOC 从故障角度出发，围绕着整个故障的全生命周期，从监控发现到故障预判，根据预判进行信息通告，相关研发人员接收到通告后进行故障定位，根据排查进展进行决策恢复，以及故障结束后的复盘及改进 action 有效性的演练验证。依靠这套故障体系，极大地降低了阿里巴巴经济体的故障风险，提升了稳定性。

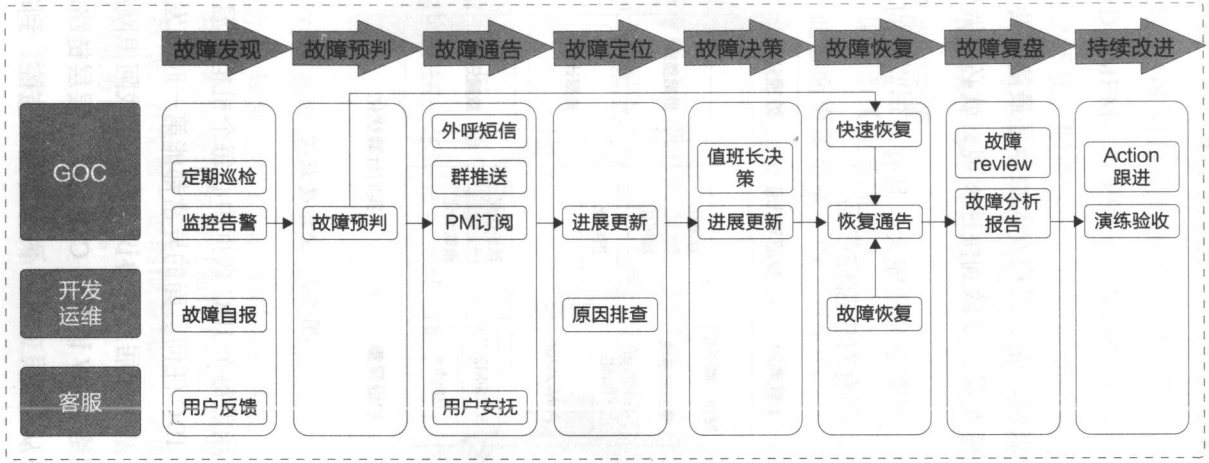


图 5-1　故障应急响应全局流程图

作为应急的关键环节，接下来针对故障发现、应急响应（包含故障预判、故障通告和故障恢复环节）、故障恢复重点展开讲解。

故障发现

对于每个故障来说，先于用户发现故障是最基本要求，监控的最主要的作用之一就是快速发现。要做到快速发现，就必须做到全维度监控。

如图 5-2 所示，阿里业务量庞大，且有多年监控的沉淀及积累，从底层 IDC 到上层业务，都有对应监控产品。

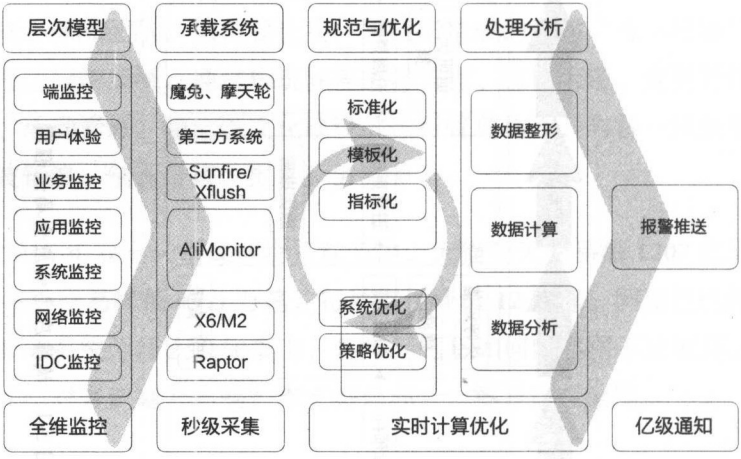


图 5-2 全维度监控

- IDC 监控：涵盖每个 IDC 数据中心每个机柜的温度、湿度、电力监控，确保了 IDC 任何层面的异常都能第一时间发现。
- 网络监控：基于邻居发现协议，自动完成阿里网络架构的绘制，从服务器到网络 ASW 设备到 CSR 设备，层层链路均是自动完成；网络设备自身从电源、CPU、内存等基本监控，到端口流量、错包率等细致监控，能涵盖所有线上运行网络设备；加上专业的 7×24 小

时网络运营现场值班团队，确保了任何网络层面的异常均能第一时间响应及处理，且根据链路均能明确受影响服务器的业务。

- 系统监控：对于 OS 层面的基础监控，阿里早已实现如 PING、SSH、CPU、LOAD、MEM、TCPRetry、DISK、Traffic 等监控指标的标准化及监控自动添加，服务器一上线即自动部署所有基础监控。
- 应用监控：对于阿里常见的应用服务监控，如 nginx、apache、jvm，包括 HSF、METAQ 等中间件监控，均有相应的标准监控模板，支持用户快速添加部署。
- 业务监控：见下节《业务监控》。
- 用户体验：此层面更多是基于第三方检测工具，从全国各地模拟访问阿里主要业务的耗时及健康度，并汇总展示用户侧的全国用户体验质量。
- 端监控：随着移动互联网的发展，对于手机客户端的监控也越来越重要，阿里已实现监控标准化，并且有成熟的监控平台支持。

业务监控

要做到故障无所遁形，每个维度的监控都必须做到精细、准确，且必须有监控报警处理的闭环，不断地去优化监控。所有监控维度中，业务监控最为关键。

业务监控在于对业务产品实际功能提供状态监控，包括正常使用状态（如交易创建的成功）、异常状态（如添加购物车失败）及质量（如详情页面打开缓慢）等，将这些业务指标通过监控埋点、数据采集、存储、计算转变为具体的监控数据，并配置一定的报警规则，从而能实时观测业务的运行状态，及时发现业务问题或故障。

业务监控相对于更底层的网络、系统、应用监控，主要有以下几方面的优势。

- 质量监控：如果将系统监控、网络监控、应用监控看作过程监控，那么业务监控就是质量监控。前面所有的系统、网络、应用的异常都会通过业务的最终情况在一定程度上体现出来。因此业务监控不仅可以发现本身系统的问题，也可以发现由于上游接口或者系统带来的问题。
- 影响面分析：通过业务监控能够较为清晰地看到整个业务的影响面，从而在故障发生的时候能够迅速地对其定级。
- 直观、易理解：业务监控的学习和理解成本较低，只要知道这个业务指标的含义就能分析出现告警时的影响情况，而其他的一些系统和应用监控则需要较专业的技术知识。

故障发现渠道

除了上文说到的通过监控发现故障，我们还有如下渠道，不做重点展开，只做简单介绍。

- 基线告警：通过算法对历史数据分析，生成更精确的基线数据，擅长发现业务指标缓慢下跌故障场景。
- 故障自报：通过代码 review、应用日志分析、数据分析等途径发现线上故障。
- 内部/外部用户反馈：用户使用时遇到的线上问题，舆情监控等。

应急响应

如何第一时间对核心业务报警做到及时、准确、有效的响应，前期需要通过工具产品做大量前置运营工作，确保在真正故障来临时的应急响应所需信息的获取成本足够的低，操作效率足够高。

响应优先级划分

- 阿里内部对于每个业务故障均有定义，并合并到《阿里巴巴集团故

障等级定义》中，GOC 的 7×24 小时专业运行值班团队会根据不同等级进行优先级区分，进行应急响应。

区分故障等级的原则为：

- 影响产品、服务、功能的重要性（核心、次核心、非核心）。
- 影响面（受影响用户数比例、PV、流量等）。
- 故障处理时间（处理时长=解决时间-发现时间）。

不同产品线、服务可依据自身的特点为每一级别设定定义，定义由技术支持召集相关角色讨论，确定版本需得到各方认同方能发布。

CMDB 建设及运营

高效率的运作离不开 CMDB 的建设及运营，对于一套完善的应急响应体系的 CMDB，应包含如下信息。

- 阿里巴巴集团故障等级定义：可以获取所有业务的故障处理优先级。
- 信息通告范围：启动应急响应后，需要进行信息通告的故障相关干系人。
- 信息通告方式：根据不同的故障等级，对应有不同的通告方式。
- 信息通告模板：符合信息通告规范的模板，能够提升应急响应时信息编写的效率及质量，有效地保证了通告出去的内容通俗易懂，信息全面。
- 监控项与故障等级的关联关系：将监控报警与故障等级定义进行衔接，相当于将报警与整个 CMDB 进行连接，确保每条高优先级故障均有完善的监控覆盖，确保每条报警的响应均有章可循。
- 监控项与快速恢复应急预案的关联关系：将监控报警与应急预案进行关联，确保发现故障的第一时间，能够执行已有预案，第一时间优先恢复故障。

整个 CMDB 的运营均由 GOC 维护，确保了信息的准确性及完整性。

应急响应流程线上化

阿里故障应急的原则为：

- 以恢复线上业务为第一要务，如执行既定预案、快速回滚变更。
- 故障排查与恢复两条腿并行走。
- 故障进展信息第一时间同步，确保所有处理人员之间故障信息、处理进展互相透明。
- 故障处理资源调度，进行必要的管理升级。

基于此原则，阿里巴巴设计并持续运作了一套应急响应流程，并且将整个流程上的所有操作进行了标准化及工具化，确保每个应急操作遵循流程指引，将因为个人能力不同而导致应急能力不同的差异性风险降到最低，最终由 7×24 小时运行值班团队基于此工具进行故障的应急响应。

快速恢复

对于当前阿里的用户体量来说，每多一秒的故障，影响的用户将会成指数倍的增加，对用户体验带来极大的损害。如何对业务进行快速恢复，有效的应急预案建设将是基础。

以阿里交易线异地多活架构为例。异地多活架构是阿里交易线根据自身业务的特点，在高可扩展性、高可用性要求下，设计的一套架构，目的是解决业务只分布在一个机房或一个城市下，难以扩展，不具备容灾能力的问题。因此，通过异地多活，可以做到：

- 需要多个跨地域的数据中心。异地多活是跨地域的，而且距离一定要做到 1000 公里以上的范围，其实在全国范围内各个城市都可以去布了。

- 每个数据中心都要承担用户的读写流量。如果只是只读业务，作用不是很大。
- 多点写。因为每个数据中心去承担用户读写流量的话，如果读或写集中到全国一个点的话，整个延迟是没有办法承受的。
- 任意一个数据中心出问题的时候，其他中心都可以分钟级去接管用户的流量。

因此，当阿里交易线异地多活建设完成后，通过业务监控发现任一数据中心出现问题时，结合 CMDB 已有的快速恢复应急预案关联关系，7×24 运行值班人员在第一时间即可知晓此故障有对应的应急预案，并基于工具进行快速恢复操作，整个故障持续时长将控制在分钟级。相比传统的故障处理，有了快速恢复之后，将跳过故障排查及定位环节，直接进行故障恢复，消除用户影响，给故障处理人充分的时间进行原因排查及问题解决，真正做到让故障无处遁形。

故障复盘及改进

故障复盘

故障处理结束后，由技术支持在 3 个工作日内召集相关人员召开故障复盘会议。如图 5-3 所示，故障复盘的流程围绕着“故障回顾、处理过程简述、故障原因分析、改进预防措施制定、故障评级、故障改进制定”来展开。

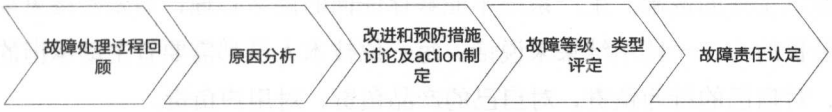


图 5-3　故障复盘流程

与会人员以故障相关处理及责任团队为主，故障复盘结束后，由技术

支持统一输出故障复盘报告，并录入系统存档。

故障防范

故障改进是紧密围绕着“如何发现故障”、“如何应急处理”、“如何彻底消除隐患”的思路输出可落地执行的改进项。

- 如何发现故障：更细颗粒度的系统监控。
- 如何应急处理：是否有降级预案、是否有自动容灾手段（如切流、限流、超时控制等）、是否有做好影响面控制（如隔离）、是否有人工容灾手段（如人工热点前置等）。
- 如何彻底消除隐患：缺陷修复、架构优化、冗余架构的设计等。
- 其他：流程机制建立、文档梳理及沉淀、知识宣讲、业务合同等。

故障改进项需明确具体的落地方案、改进项责任人、计划完成时间、校验标准；故障改进项在故障复盘现场达成一致后，由改进项责任人定期更新改进项完成进展情况，技术支持定期提供改进项完成情况报告。

小结

阿里巴巴的首席技术官行癫于阿里巴巴 2017 年的技术大会中提出“能避免的故障尽量避免，不能避免的故障快速恢复”。对于阿里这样体量的公司，系统的稳定性影响的不只是用户的体验，更是许多行业能正常运转甚至赖以生存的基础。生产系统不能够存任何的侥幸心理，需要始终秉承一颗敬畏的心，一切用结果来说话。每一位技术人员都需要有主动承担的精神，对自己的行为负责，对自己的产品负责，对用户负责。

5.2　体系化思考，高效解决运营商问题

上篇文章，介绍了阿里巴巴集团完整的故障管理体系，该故障管理体系已经持续运作多年，是一个较完备的体系。但是这个体系也是在实战中不断打磨而成，且在不同故障场景下进行灵活运用，本文将结合运营商类型故障介绍一下演进过程。

背景

在应对第三方运营商的故障过程中，运行管理人员早先建立起的故障管理体系，大致如图 5-4 所示。

流程中有四个角色：GOC（Global Operations Center）全球运行指挥中心、GNOC（Global Network Operations Center）全球网络运行指挥中心、售后（即客服）、用户。与上篇文章的故障管理体系相比，多了一个 GNOC 的角色。GOC 和 GNOC 均为阿里巴巴承担 7×24 小时线上业务的运行管理域团队。从上图流程也可以发现，两个团队主要承担故障发现、应急处理、信息流转等运行管理域工作。

售后主要负责对接用户，解决用户反馈的个例问题，并对批量的反馈向上汇报。

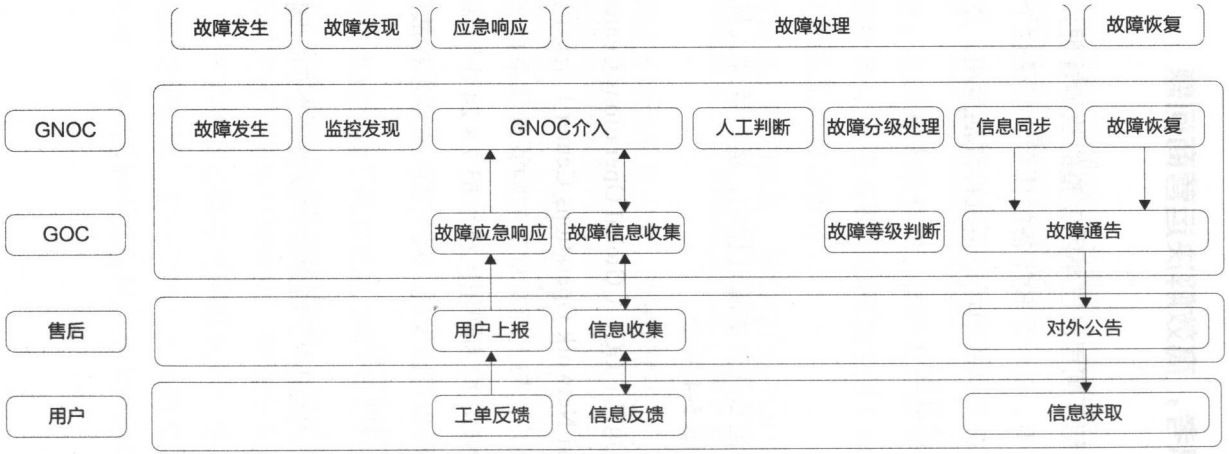


图 5-4　运营商故障管理体系 V1.0

问题现象

某日正在日常工作的网络人员收到售后人员的反馈：多例客户投诉访问香港阿里云异常，怀疑阿里云在香港机房的网络可能存在问题，需要紧急处理，恢复客户业务。

经过排查，网络并未发现香港机房存在异常，怀疑某运营商的链路存在异常，但是具体验证需要通过售后拿到客户的双向 MTR 信息来判断具体的故障点在哪儿，这样才能向具体运营商做紧急报障处理。此时，网络人员需要售后从反馈用户处获取到更详细的信息。而用户回复信息的不可控，时间一长，往往会导致用户反馈量迅速增加，进而导致网络人员和售后人员都来回奔波，疲于应对。

问题剖析

围绕整个运营商故障的生命周期，无论是从运营商故障发生，故障监控发现，GNOC 应急响应并介入处理，根据相关具体信息进行排查判断，依照固有的故障分级进行操作处理，并及时跟相关人员进行信息同步，直至故障恢复。依靠这套故障体系，极大地地降低了阿里巴巴经济体的故障风险，提升了稳定性。

通过背景介绍，我们可以看到，已有的应急体系相对较为完备。那么，为什么在运营商类故障中同样的流程会效率较低呢？

分析运营商故障处理流程的五个阶段，我们发现最主要的问题是故障处理阶段强依赖于用户的工单反馈和信息反馈，从而导致类似故障处理时长不可控，具体来说，主要有如下两点。

（1）运营商故障处理被动，故障是因为外部因素导致，而发现来源主要依赖于客户工单上报，根据工单数量来评估影响及分级处理故障，导致

类似故障的处理时长不可控。

（2）故障处理过程中涉网络、售后、客户等多种角色，且有外部用户，沟通成本较大。

解决方案

根据上述问题剖析，我们可以发现时间最不可控的部分为交错在GNOC、GOC、售后和用户多个角色之间的信息互动流程，因此需要通过解绑对用户的强依赖，来对整个运营商故障的运行管理域流程进行优化，从而有效缩短故障处理时长，从而降低故障影响。

首先在运营商故障的发现能力上，要做到脱离用户反馈的精准和实时发现。然而具备这个能力并不容易，我们大致经历了全手工、第三方工具、工具化、产品化四个主要阶段。具体有如下几点。

（1）推动互联网监控能力（区别于内网监控）的优化，一是扩大覆盖面，保障国内全部城市及海外重点国家全覆盖；二是聚焦关键链路，对国内主要运营商，核心城市群做重点探测布点和监控优化；三是提高监控实时性，推动对任务分发、探测协议、数据传输链路耗时进行严格把控；四是推动算法团队对监控算法的优化，使其更精确，保证每次能快速准确判断故障点。

（2）推动运营商故障容灾，实现自动化、智能化，当监控发现某城网某运营商网络异常并判断是运营商故障时，不再是仅仅密切关注，而是准确判断、立即处理，通过系统自动执行容灾切换，保证业务能第一时间恢复。具体实现方案第二章第三篇文章做过介绍，在此就不一一赘述。

（3）推动异常信息获取自动化，当互联网监控发现异常时自动触发链路跟踪机制，主动获取MTR、TCP ping、DNS等信息，并自动收集测试信

息判断故障点，第一时间报障运营商处理，并主动通知售后人员，既无需等客户上报并提供测试结果后再报障，大大缩短处理时间，同时也可以使售后在对接客户时更为主动，提升用户体验。

其次是在运营商故障处理的信息流转上，要做到主动并及时通知，方能更好解绑对用户信息的强依赖。然而做到这一点改变也很不容易，运行管理团队为之付出了很多的努力，主要有如下几点。

（1）完善互联网故障处理体系，一是提高运行管理域团队对运营商网络故障的感知度；二是信息及时同步，当监控互联网有略微异常时，能尽快同步信息口径给到各个角色，尤其是售后人员，赋能其在面对用户时有更多实时有用的信息口径；三是预先对运营商各类故障的潜在影响做完整评估，并完善相应的故障处理预案，有的放矢。

（2）同国内主要运营商深入合作，通过钉钉、电话、邮件等多种沟通方式，协同实现运营商故障的快速响应、快速定位和快速处理。因为有特殊合作方式，在运营商故障处理上，阿里巴巴变得更为敏捷，多次出现当阿里巴巴挂出运营商故障的公告时，多数用户还未察觉，自然也不必再费力去提工单。

（3）建立运营商日常割接的分级预案机制，实时关注运营商日常割接信息，并做好全面评估，根据评估出的具体影响程度，依据分级预案，进行不同处理。一是评估判断对业务有实际影响，且在必要情况下提前做容灾切换，保证客户业务的稳定运行，防患于未然；二是若切换有损，在切换前会尽早对外发布公告，并明确告知具体影响，为用户争取时间，减轻业务影响。

推动并落地上述措施，最终形成完备流程并迭代沉淀为自动化产品工具，逐渐解绑了对用户信息的强依赖，使整个流程变得高效且可控。优化过后的具体流程，大致如图 5-5 所示。

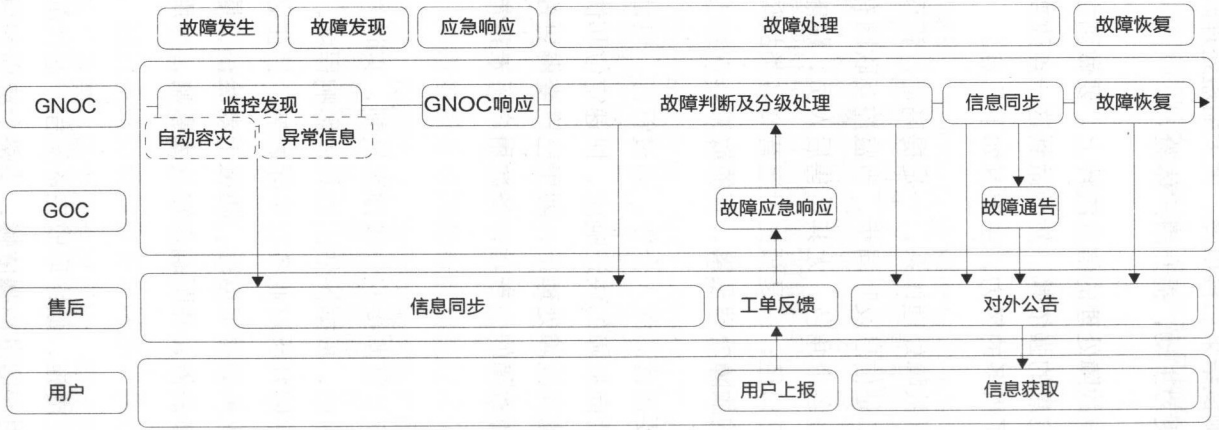


图 5-5　运营商故障管理体系 V2.0

小结

随着运行管理域流程的逐步完善，在应用运营商故障中，我们真正做到了“先于用户发现，自动报障公告”。

现在当发生运营商网络故障时，网络人员会收到异常告警，然后只需要依据系统获取的 MTR 信息便可以立即向运营商报障，并告知售后人员实时关注相关工单反馈状况并挂出对外公告，如果故障升级则自动执行容灾切换。整个流程顺畅无阻，没有多角色来回沟通，大大提升了用户体验和处理效率。

本文仅列举了运营商故障做运行管理域演进的案例，这种模式可以应用到更多的内外部协同处理的流程中，任何好的运行管理体系都需要灵活运用并结合业务特性进行优化不断演进，广大读者可以灵活运用。

5.3　以战养兵，以故障演练提升系统稳定性

背景

阿里巴巴经过多年的技术演进，系统工具和架构已经高度垂直化，服务器规模也达到了比较大的体量。当服务规模大于 10000 台时，小概率的硬件故障每天都会发生。这时如果需要人的干预，系统就无法可靠的伸缩。为此每一层的系统都会面向失败做设计，对下游组件零信任，确保在故障发生时可以快速地发现和处理。

不过这些措施在故障发生时是否真的有效？恢复故障的工具是否实现了容灾？处理问题的人是否熟练？沟通机制是否有疏漏？容灾措施的影响是否会辐射到上一层？这些问题，平时没有太多的机会验证，只能在真实

故障中暴露。

故障演练就在这个背景下诞生，沉淀通用的故障场景，以可控成本在线上将故障重现，以持续性的演练和回归方式的运营来暴露问题，不断验证和推动系统、工具、流程、人员能力的提升，从而提前发现并修复可避免的重大问题，同时通过演练的不断打磨达到缩短故障修复时长的目的。

一次生产环境故障复现的案例

2016 年某日，某业务开始出现大量异常告警，影响持续若干分钟。原因是该业务依赖的一个持久化服务因为机器磁盘有问题（是一个驱动的 bug），工作线程卡住了，但机器心跳线程正常，导致负载中心无法摘除异常机器，使得业务请求到这台机器的所有请求发生超时，从而造成业务下跌。

对于该问题的改进方案是，让持久化服务的工作线程来发送心跳，当工作线程异常的时候，不再发送心跳包，负载中心就可以把问题机器摘掉。此外，该持久化服务也重新梳理了相关的出错处理逻辑，以应对网络抖动的问题。

在 B 完成改进方案后，6 月份技术人员在线上进行了第一次故障重现，模拟的故障场景就是磁盘无法读写。演练过程持续了 62s，业务的 RT 稍有波动，不过业务没有下跌服务成功完成容灾。

虽然这次故障演练的场景非常简单，不过在故障治理的历史上将具有非常重要的意义，因为第一次在真实环境下实现了故障修复和验证的闭环。

故障画像分析和演练模型设计

阿里巴巴因为其多元化的业务场景和日益复杂的技术架构，会遇到各

式各样的故障，如果对故障整体做初步画像，故障整体可以分为 IaaS 层、PaaS 层、SaaS 层的故障（如图 5-6 所示）。

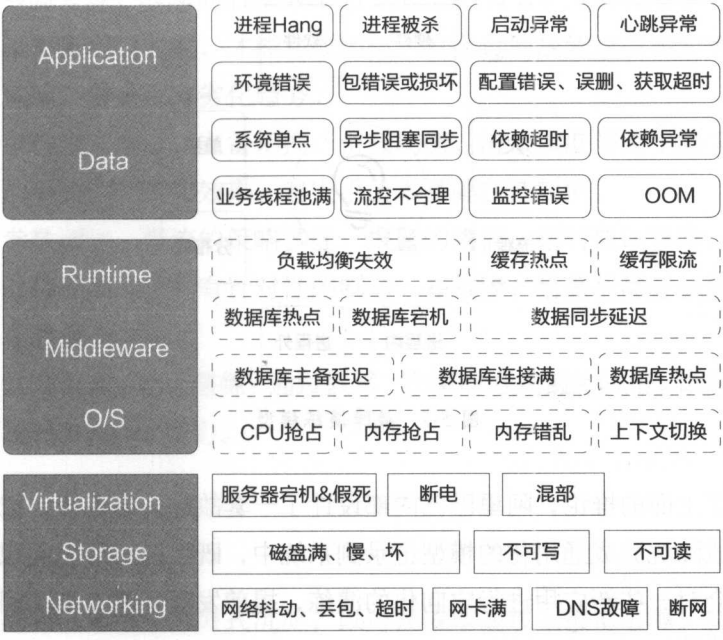


图 5-6　故障画像分析

在真实的故障复现和演练中，我们需要一个高度抽象的模型。所以我们把故障模型又做了一次升级，并得到一些推论（如图 5-7 所示）如下。

（1）任何故障，一定是硬件如 IaaS 层，软件如 PaaS 或 SaaS 的故障。并且有个规律，硬件故障的现象，一定可以在软件故障现象上有所体现。

（2）故障一定隶属于单机或是分布式系统之一，分布式故障包含单机故障。对于单机或同机型的故障，以系统为视角，故障可能是当前进程内的故障，比如 FullGC，CPU 飙高；也可能是进程外的故障，比如其他进程突然抢占了内存，导致当前系统异常等。大多数时候，我们只关注故障对当前系统的影响，而不是真的需要外部产生故障。

同时，还可能有一类故障，可能是人为失误，或流程不适当导致，这部分不做重点讨论。

图 5-7 故障演练模型

有了上面的推论，阿里巴巴内部设计了一套故障演练系统，把历史发生过的故障沉淀成通用化的模型记录到系统中，既可以对历史问题进行回放，也会对一些新应用进行定向化的演练，提前发现问题，推动问题修复和架构改造。

故障演练的一些实践

在 2016 年双 11 中，线上故障演练第一次成规模、有组织地开展起来。演练平台和演练经验在多个阿里子公司中复制和传播，故障工具也大幅度缩减了业务方演练的成本。有近十个 BU 参与，数百个演练场景设计，数十次大大小小的演习，发现并解决了大量的问题。故障演练主要应用在下面几个场景。

- 预案有效性：过去的预案测试的时候，线上没有问题，所以就算测试结果符合预期，也有可能是有意外但是现象被掩藏了。
- 监控报警：报警的有无、提示消息是否准确、报警实效是 5 分钟还

是半小时、收报警的人是否转岗、手机是否欠费等，都是可以 check 的点。

- 故障复现：故障的后续 Action 是否真的有效，完成质量如何，只有真实重现和验证，才能完成闭环。发生过的故障也应该时常拉出来练练，看是否有劣化趋势。
- 架构容灾测试：主备切换、负载均衡，流量调度等为了容灾而存在的手段的时效和效果，容灾手段本身健壮性如何。
- 参数调优：限流的策略调优、报警的阈值、超时值设置等。
- 故障模型训练：有针对性的制造一些故障，给做故障定位的系统制造数据。
- 故障突袭、联合演练：通过蓝军、红军的方式锻炼队伍，以战养兵，提升 DevOps 能力。

小结

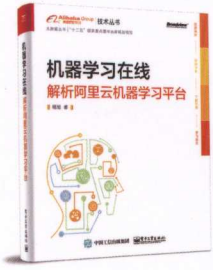
把故障以场景化的方式沉淀，以可控成本在线上模拟故障，让系统和工程师平时有更多实战机会，加速系统、工具、流程、人员的进步。

故障演练的目的在于：演练常态化、故障标类化、演练智能化。用常态化的演练驱动稳定性进步，而不是大促前进行补习；丰富更多的故障场景，定义好最小故障场景和处理手段；基于架构和业务分析的智能化演练，沉淀行业故障演练解决方案。

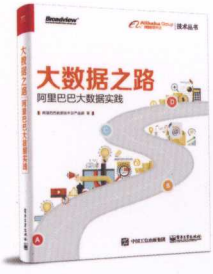
好书分享

《尽在双11：阿里巴巴技术演进与超越》
ISBN 978-7-121-30917-5
阿里巴巴集团双11技术团队 著
精炼揭秘双11八年技术演进史

《技术之瞳：阿里巴巴技术笔试心得》
ISBN 978-7-121-29933-9
阿里巴巴集团校园招聘笔试项目组 著
进击阿里巴巴技术岗的精准指南



《机器学习在线：解析阿里云机器学习平台》
ISBN 978-7-121-31869-6
杨旭 著
永不掉线的机器学习实践库，助你快速掌握AI技术！



《大数据之路：阿里巴巴大数据实践》
ISBN 978-7-121-31438-4
阿里巴巴数据技术及产品部 著
阿里巴巴分享对大数据的认知、与世界共创数据智能的重要基石。

在钉钉上扫一扫加我

反馈意见或投稿
邮箱：dongx@phei.com.cn
微信：JK--DX

Broadview® www.broadview.com.cn 博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

这本《逆流而上：阿里巴巴技术成长之路》总结了近年来阿里巴巴集团在重要领域中遇到的故障和排查方法。从故障中得到的教训，剖析出来的架构缺陷，折射出来的实现问题以及运维过程中的疏忽和错误都是很真实的，很具有说服力。他山之石，可以攻玉，希望为广大开发和运维工程师带来帮助。

—— 阿里云资深总监 吴结生

古语说：前人栽树，后人乘凉。本书创作旨在让历史的经验传承下去，帮助到更多的人。它汇集了阿里巴巴集团各个 BU 技术人员在日常工作中所遇到的典型踩坑案例，这些案例全部来自线上生产实践，涉及基础设施、中间件、数据库、业务开发以及稳定性建设，基本涵盖了阿里巴巴所有的技术兵种，这是一本非常全面的技术踩坑实践书，具有很重要的参考意义。

—— 阿里云研究员 褚霸

阿里巴巴的技术人员日常的研发运维过程，就是不断和新问题斗智斗勇的过程，我们会鼓励把遇到的挑战和问题总结出来，所以在这个过程中积累了大量的总结资料，这些资料有些总结到了产品里，成为架构、系统的一部分，有些不断被学习变成了其他更多同事的新能力。

—— 中间件技术部研究员 小邪

在我带领阿里巴巴 GOC（全球运行指挥中心）团队期间，天天面对不断发生的大小故障，尤其是重复发生的故障。而此书恰恰是在这种思考之下所采取的行动之一。成功难以模仿，教训可以学习。每篇文章的背后都是血淋淋的教训，值得每一个技术人员好好阅读。

—— 菜鸟资深专家 王乐

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：董　雪
责任编辑：林瑞和
封面设计：吴海燕

上架建议：计算机

ISBN 978-7-121-32768-1

定价：59.00元